USENIX

Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems — *Seattle, WA, Mar. 26–28, 2003*

conference

proceedings

4th USENIX Symposium on Internet Technologies and Systems

Seattle, Washington, USA
March 26–28, 2003

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-931971-08-0

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)

March 26–28, 2003
Seattle, WA, USA

Symposium Organizers

Program Chair

Steven D. Gribble, *University of Washington*

Program Committee

Mike Dahlin, *University of Texas at Austin*
Armando Fox, *Stanford University*
Peter Honeyman, *CITI, University of Michigan*
Terence Kelly, *University of Michigan*
Hank Levy, *University of Washington*
Vivek Pai, *Princeton University*
Srinivasan Seshan, *Carnegie Mellon University*
Amin Vahdat, *Duke University*
Geoff Voelker, *University of California, San Diego*
Dan Wallach, *Rice University*

The USENIX Association Staff

External Reviewers

Mukesh Agrawal
Leeann Bent
Ranjita Bhagwan
Ashwin Bharambe
Nikita Borisov
Aaron Brown
Yang Hua Chu
Jim Gray
Krishna Gummadi
Andy Huang
Zachary Ives
Emre Kiciman
Dejan Kostic
Ben Ling
Kevin Lochner
David Moore
Suman Nath
Shankar Ponnekanti
Niels Provos
Lili Qiu
Sanjay Rao
Dan Reeves
Mema Roussopoulos
Stefan Saroiu
Michael Swift
Kiran Tati
Renata Teixeira
Matt Welsh
Andrew Whitaker
Alec Wolman
Jung Yang
Ken Yocum
Haifeng Yu

USITS '03: 4th USENIX Symposium on Internet Technologies and Systems

March 26–28, 2003
Seattle, WA, USA

Index of Authors

Message from the Symposium Chair

It is my pleasure to welcome you to USITS '03, the 4th USENIX Symposium on Internet Technologies and Systems.

The Internet has continued to evolve in unexpected ways. I believe this year's program captures the excitement and innovation that our community demonstrates in the research and development of Internet systems. We will hear presentations that span a range of timely and important topics, such as security, Internet robustness, novel distributed systems and architectures, and performance.

USITS '03 received 75 submissions, each of which was reviewed by several program committee members and, as appropriate, external reviewers. The program committee was very conservative in its approach to addressing conflicts: PC members did not participate in the discussion of any papers co-authored by themselves, or anyone who has or might be perceived to have a conflict of interest. After a ten hour meeting, we selected 19 high quality papers for inclusion in the symposium. The program committee maintained a high bar for acceptance, and as a result, we are uniformly pleased with the set of papers in this year's proceedings.

I would like to thank the program committee for their high quality work and dedication, without which we could not have selected such a strong program. I would also like to thank the many external reviewers: your expertise and rapid turnaround was greatly appreciated. Finally, I am indebted to the USENIX staff whose professionalism and efforts made this symposium possible.

Steven D. Gribble, ***University of Washington***
Program Chair

Why do Internet services fail, and what can be done about it?

David Oppenheimer, Archana Ganapathi, and David A. Patterson
University of California at Berkeley, EECS Computer Science Division
387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA
{davidopp,archanag,patterson}@cs.berkeley.edu

Abstract

In 1986 Jim Gray published his landmark study of the causes of failures of Tandem systems and the techniques Tandem used to prevent such failures [6]. Seventeen years later, Internet services have replaced fault-tolerant servers as the new kid on the 24x7-availability block. Using data from three large-scale Internet services, we analyzed the causes of their failures and the (potential) effectiveness of various techniques for preventing and mitigating service failure. We find that (1) operator error is the largest cause of failures in two of the three services, (2) operator error is the largest contributor to time to repair in two of the three services, (3) configuration errors are the largest category of operator errors, (4) failures in custom-written front-end software are significant, and (5) more extensive online testing and more thoroughly exposing and detecting component failures would reduce failure rates in at least one service. Qualitatively we find that improvement in the maintenance tools and systems used by service operations staff would decrease time to diagnose and repair problems.

1. Introduction

The number and popularity of large-scale Internet services such as Google, MSN, and Yahoo! have grown significantly in recent years. Such services are poised to increase further in importance as they become the repository for data in ubiquitous computing systems and the platform upon which new global-scale services and applications are built. These services' large scale and need for 24x7 operation have led their designers to incorporate a number of techniques for achieving high availability. Nonetheless, failures still occur.

Although the architects and operators of these services might see such problems as failures on their part, these system failures provide important lessons for the systems community about why large-scale systems fail, and what techniques could prevent failures. In an attempt to answer the question "Why do Internet services fail, and what can be done about it?" we have studied over a hundred post-mortem reports of user-visible failures from three large-scale Internet services. In this paper we

- identify which service components are most failure-prone and have the highest Time to Repair (TTR), so that service operators and researchers can know what areas most need improvement;
- discuss in detail several instructive failure case studies;
- examine the applicability of a number of failure mitigation techniques to the actual failures we studied; and
- highlight the need for improved operator tools and systems, collection of industry-wide failure data, and creation of service-level benchmarks.

The remainder of this paper is organized as follows. In Section 2 we describe the three services we analyzed and our study's methodology. Section 3 analyzes the causes and Times to Repair of the component and service failures we examined. Section 4 assesses the applicability of a variety of failure mitigation techniques to the actual failures observed in one of the services. In Section 5 we present case studies that highlight interesting failure causes. Section 6 discusses qualitative observations we make from our data, Section 7 surveys related work, and in Section 8 we conclude.

2. Survey services and methodology

We studied a mature online service/Internet portal (*Online*), a bleeding-edge global content hosting service (*Content*), and a mature read-mostly Internet service (*ReadMostly*). Physically, all of these services are housed in geographically distributed colocation facilities and use commodity hardware and networks. Architecturally, each site is built from a load-balancing tier, a stateless front-end tier, and a back-end tier that stores persistent data. Load balancing among geographically distributed sites for performance and availability is achieved using DNS redirection in *ReadMostly* and using client cooperation in *Online* and *Content*.

Front-end nodes are those initially contacted by clients, as well as the client proxy nodes used by *Content*. Using this definition, front-end nodes do not store per-

service characteristic	Online	ReadMostly	Content
hits per day	~100 million	~100 million	~7 million
# of machines	~500, 2 sites	> 2000, 4 sites	~500, ~15 sites
front-end node architecture	Solaris on SPARC and x86	open-source OS on x86	open-source OS on x86
beck-end node architecture	Network Appliance filers	open-source OS on x86	open-source OS on x86
period of data studied	7 months	6 months	3 months
component failures	296	N/A	205
service failures	40	21	56

Table 1: Differentiating characteristics of the services described in this study.

sistent data, although they may cache or temporarily queue data. *Back-end nodes* store persistent data. The "business logic" of traditional three-tier system terminology is part of our definition of front-end, because these services integrate their service logic with the code that receives and replies to client requests.

The front-end tier is responsible primarily for locating data on back-end machine(s) and routing it to and from clients in *Content* and *ReadMostly*, and for providing online services such as email, newsgroups, and a web proxy in *Online*. In *Content* the "front-end" includes not only software running at the colocation sites, but also client proxy software running on hardware provided and operated by *Content* that is physically located at customer sites. Thus *Content* is geographically distributed not only among the four colocation centers, but also at about a dozen customer sites. The front-end software at all three sites is custom-written, and at *ReadMostly* and *Content* the back-end software is as well. Figure 1, Figure 2, and Figure 3 show the service architectures of *Content*, *Online*, and *ReadMostly*, respectively.

Operationally, all three services use primarily custom-written software to administer the service; they undergo frequent software upgrades and configuration updates; and they operate their own 24x7 System Operations Centers staffed by operators who monitor the service and respond to problems. Table 1 lists the primary characteristics that differentiate the services. More details on the architecture and operational practices of these services can be found in [17].

Because we are interested in *why* and *how* large-scale Internet services fail, we studied individual problem reports rather than aggregate availability statistics. The operations staff of all three services use problem-tracking databases to record information about component and service failures. Two of the services (*Online* and *Content*) gave us access to these databases, and one of the services (*ReadMostly*) gave us access to the problem post-mortem reports written after every major user-visible service failure. For *Online* and *Content*, we defined a user-visible failure (which we call a *service failure*) as one that theoretically prevents an end-user from accessing the service or a part of the service (even if the user is given a reasonable error message) or that significantly degrades a user-visible aspect of system performance[1]. Service failures are caused by component failures that are not masked.

Our base dataset consisted of 296 reports of component failures from *Online* and 205 component failures from *Content*. These component failures turned into 40 service failures in *Online* and 56 service failures in *Content*. *ReadMostly* supplied us with 21 service failures (and two additional failures that we considered to be

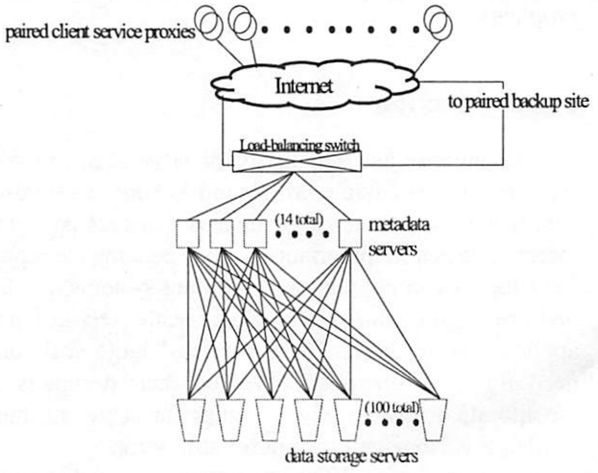


Figure 1: The architecture of one site of *Content*. Stateless metadata servers provide file metadata and route requests to the appropriate data storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over UDP. Each cluster is connected to its twin site via the Internet.

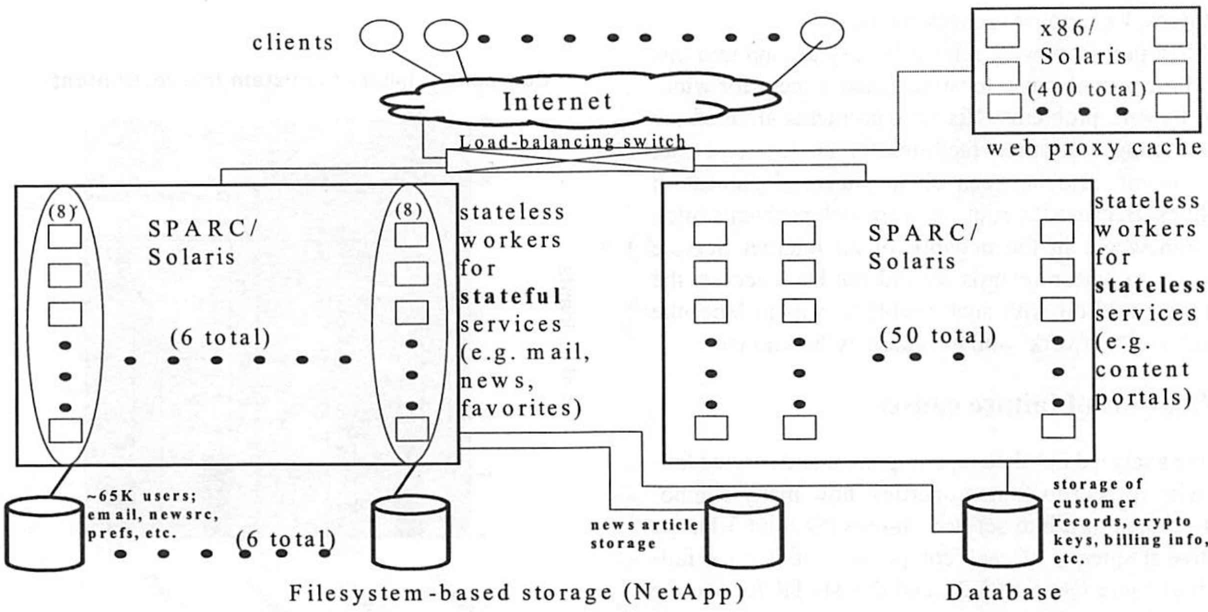

clients
Internet
Load-balancing switch
x86/ Solaris (400 total)
web proxy cache
SPARC/ Solaris
(8)
(6 total)
stateless workers for stateful services (e.g. mail, news, favorites)
SPARC/ Solaris
(50 total)
stateless workers for stateless services (e.g. content portals)
~65K users; email, newsrc, prefs, etc.
(6 total)
news article storage
storage of customer records, crypto keys, billing info, etc.
Filesystem-based storage (NetApp)
Database

Figure 2: The architecture of one site of *Online*. Depending on the particular feature a user selects, the request is routed to any one of the web proxy cache servers, any one of 50 servers for stateless services, or any one of eight servers from a user's "service group" (a partition of one sixth of all users of the service, each with its own back-end data storage server). Persistent state is stored on Network Appliance servers and is accessed by worker nodes via NFS over UDP. This site is connected to a second site, at a collocation facility, via a leased network connection.

below the threshold to be deemed a service failure). These problems corresponded to 7 months at *Online*, 6 months at *ReadMostly*, and 3 months at *Content*. In classifying problems, we considered operators to be a component of the system; when they fail, their failure may or may not result in a service failure.

We attributed the cause of a service failure to the first component that failed in the chain of events leading up to the service failure. The *cause* of the component failure was categorized as node hardware, network hardware, node software, network software (*e.g.*, router or switch firmware), environment (*e.g.*, power failure), operator error, overload, or unknown. The *location* of that component was categorized as front-end node, back-end node, network, or unknown. Note that the underlying flaw may have remained latent for some time, only to cause a component to fail when the component was used in a particular way for the first time. Due to inconsistencies across the three services as to how or whether security incidents (*e.g.*, break-ins and denial of service attacks) were recorded in the problem tracking

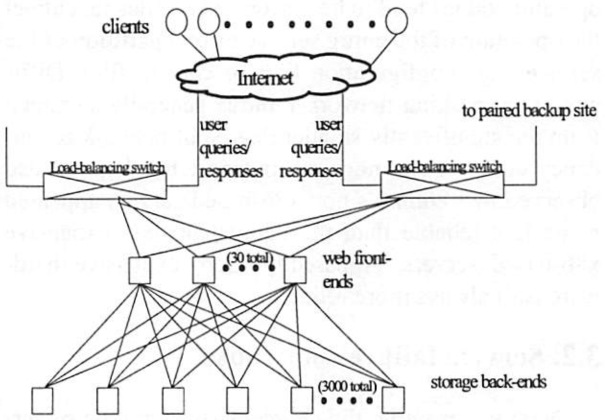


Figure 3: The architecture of one site of *ReadMostly*. A small number of web front-ends direct requests to the appropriate back-end storage servers. Persistent state is stored on commodity PC-based storage servers and is accessed via a custom protocol over TCP. A redundant pair of network switches connects the cluster to the Internet and to a twin site via a leased network connection.

[1] "Significantly degrades a user-visible aspect of system performance" is admittedly a vaguely-defined metric. It would be preferable to correlate failure reports with degradation in some aspect of user-observed Quality of Service, such as response time, but we did not have access to an archive of such metrics for these services. Note that even if a service measures and archives response times, such data is not guaranteed to detect all user-visible failures, due to the periodicity and placement in the network of the probes. In sum, our definition of *user-visible* is problems that were *potentially* user-visible, *i.e.*, visible if a user tried to access the service during the failure.

databases, we ignored security incidents.

Most problems were relatively easy to map into this two-dimensional cause-location space, except for wide-area network problems. Network problems affected the links among colocation facilities for all services, and, for *Content*, also between client sites and colocation facilities. Because the root cause of such problems often lay somewhere in the network of an Internet Service Provider to whose records we did not have access, the best we could do with such problems was to label the location as "network" and the cause as "unknown."

3. Analysis of failure causes

We analyzed our data on component and service failure with respect to four properties: how many component failures turn into service failures (Section 3.1); the relative frequency of each component and service failure root cause (Section 3.2); and the MTTR for service failures (Section 3.3).

3.1. Component failures to service failures

The services we studied all use redundancy in an attempt to mask component failures. That is, they try to prevent component failures from turning into end-user visible failures. As indicated by Figure 4 and Figure 5, this technique generally does a good job of preventing hardware, software, and network component failures from turning into service failures, but it is much less effective at masking operator failures. A qualitative analysis of the failure data suggests that this is because operator actions tend to be performed on files that affect the operation of the entire service or of a partition of the service, *e.g.,* configuration files or content files. Difficulties in masking network failures generally stemmed from the significantly smaller degree of network redundancy compared to node redundancy. Finally, we also observed that *Online*'s non-x86-based servers appeared to be less reliable than the equivalent, less expensive x86-based servers. Apparently more expensive hardware isn't always more reliable.

3.2. Service failure root cause

Next we examine the source and magnitude of service failures, categorized by the root cause location and component type. We augmented the data set presented in the previous section by examining five more months of data from *Online*, yielding 21 additional service failures, thus bringing our total to 61 for that service. (We did not analyze the component failures that did not turn into service failures from these five extra months, hence

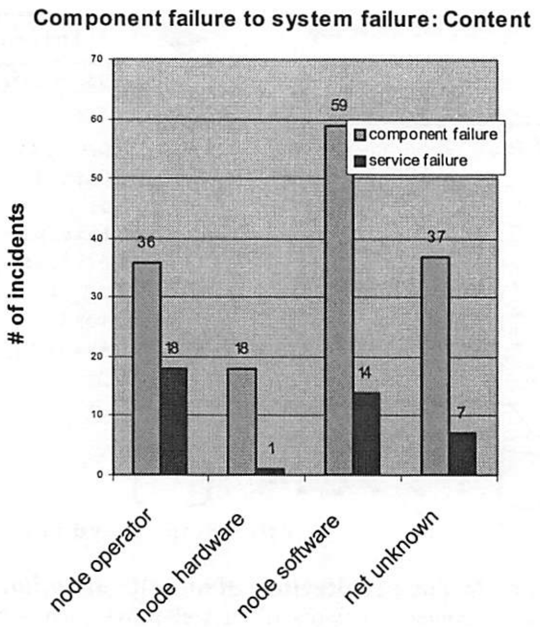


Figure 4: Number of component failures and resulting service failures for *Content*. Only those categories for which we classified at least six component failures (operator error related to node operation, node hardware failure, node software failure, and network failure of unknown cause) are listed. The vast majority of network failures in *Content* were of unknown cause because most network failures were problems with Internet connections between colocation facilities or between customer proxy sites and colocation facilities. For all but the "node operator" case, 24% or fewer component failures became service failures. Fully half of the 36 operator errors resulted in service failure, suggesting that operator errors are significantly more difficult to mask using the service's existing redundancy mechanisms.

their exclusion from Section 3.1.)

Table 2 shows that contrary to conventional wisdom, front-end machines are a significant source of failure--in fact, they are responsible for more than half of the service failures in *Online* and *Content*. This fact was largely due to operator configuration errors at the application or operating system level. Almost all of the problems in *ReadMostly* were network-related; we attribute this to simpler and better-tested application software at that service, fewer changes made to the service on a day-to-day basis, and a higher degree of node redundancy than is used at *Online* and *Content*.

Table 3 shows that operator error is the leading cause of service failure in two of the three services.

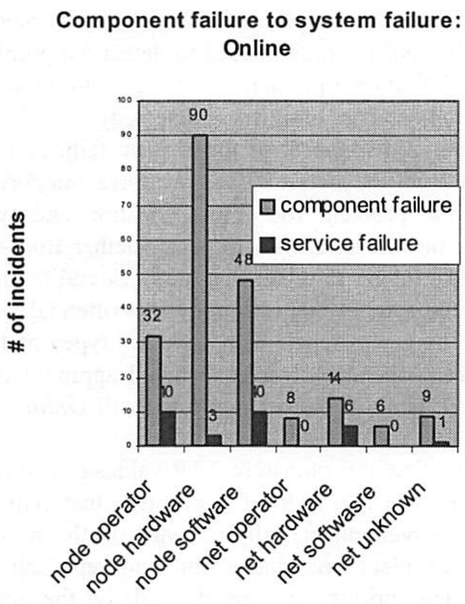


Figure 5: Number of component failures and resulting service failures for *Online*. Only those categories for which we classified at least six component failures (operator error related to node operation, node hardware failure, node software failure, and various types of network failure) are listed. As with *Content*, operator error was difficult to mask using the service's existing redundancy schemes. Unlike at *Content*, a significant percentage of network hardware failures became service failures. There is no single explanation for this, as the customer-impacting network hardware problems affected various pieces of equipment.

Operator error in all three services generally took the form of misconfiguration rather than procedural errors (*e.g.*, moving a user to the wrong fileserver). Indeed, for all three services, more than 50% (and in one case nearly 100%) of the operator errors that led to service

	Front-end	Back-end	Net-work	Un-known
Online	77%	3%	18%	2%
Content	66%	11%	18%	4%
Read-Mostly	0%	10%	81%	9%

Table 2: Service failure cause by location. Contrary to conventional wisdom, most failure root causes were components in the service front-end.

failures were configuration errors. In general, operator errors arose when operators were making changes to the system, *e.g.*, scaling or replacing hardware, or deploying or upgrading software. A few failures were caused by operator errors during the process of fixing another problem, but those were in the minority--most operator errors, at least those recorded in the problem tracking databases, arose during normal maintenance.

Networking problems were a significant cause of failure in all three services, and they caused a surprising 76% of all service failures at *ReadMostly*. As mentioned in Section 3.1, network failures are less often masked than are node hardware or software failures. An important reason for this fact is that networks are often a single point of failure, with services rarely using redundant network paths and equipment within a single site. Also, consolidation in the collocation and network provider industries has increased the likelihood that "redundant" network links out of a collocation facility will actually share a physical link fairly close (in terms of Internet topology) to the data center. A second reason why networking problems are difficult to mask is that their failure modes tend to be complex: networking hardware and software can fail outright or more gradually, *e.g.*, become overloaded and start dropping packets. Combined with the inherent redundancy of the Internet, these

	Operator node	Operator net	H/W node	H/W net	S/W node	S/W net	Unknown node	Unknown net	Environ ment
Online	31%	2%	10%	15%	25%	2%	7%	3%	0%
Con-tent	32%	4%	2%	2%	25%	0%	18%	13%	0%
Read-Mostly	5%	14%	0%	10%	5%	19%	0%	33%	0%

Table 3: Service failure cause by component and type of cause. The component is described as node or network, and failure cause is described as operator error, hardware, software, unknown, or environment. We excluded the "overload" category because of the very small number of failures caused.

failure modes generally lead to increased latency and decreased throughput, often experienced intermittently--far from the "fail stop" behavior that high-reliability hardware and software components aim to achieve [6].

Colocation facilities were effective in eliminating "environmental" problems--no environmental problems, such as power failure or overheating, led to service failure (one power failure did occur, but geographic redundancy saved the day). We also observed that overload (due to non-malicious causes) was insignificant.

Comparing this service failure data to our data on component failures in Section 3.1, we note that as with service failures, component failures arise primarily in the front-end. However, hardware and/or software problems dominate operator error in terms of component failure causes. It is therefore not the case that operator error is more frequent than hardware or software problems, just that it is less frequently masked and therefore more often results in a service failure.

Finally, we note that we would have been able to learn more about the detailed causes of software and hardware failures if we had been able to examine the individual component system logs and the services' software bug tracking databases. For example, we would have been able to break down software failures between operating system *vs.* application and off-the-shelf *vs.* custom-written, and to have determined the specific coding errors that led to software bugs. In many cases the operations problem tracking database entries did not provide sufficient detail to make such classifications, and therefore we did not attempt to do so.

3.3. Service failure time to repair

We next analyze the average Time to Repair (TTR) for service failures, which we define as the time from problem detection to restoration of the service to its pre-failure Quality of Service[1]. Thus for problems that are repaired by rebooting or restarting a component, the TTR is the time from detection of the problem until the reboot is complete. For problems that are repaired by replacing a failed component (*e.g.*, a dead network switch or disk drive), it is the time from detection of the problem until the component has been replaced with a functioning one. For problems that "break" a service functionally and that cannot be solved by rebooting (*e.g.*, an operator configuration error or a non-transient software bug), it is the time until the error is corrected, or until a workaround is put into place, whichever happens first. Note that our TTR incorporates both the time needed to diagnose the problem and the time needed to repair it, but not the time needed to detect the problem (since by definition a problem did not go into the problem tracking database until it was detected).

We analyzed a subset of the service failures from Section 3.2 with respect to TTR. We have categorized TTR by the problem root cause location and type. Table 4 is inconclusive with respect whether front-end failures take longer to repair than do back-end failures. Table 5 demonstrates that operator errors often take significantly longer to repair than do other types of failures; indeed, operator error contributed approximately 75% of all Time to Repair hours in both *Online* and *Content*.

We note that, unfortunately, TTR values can be misleading because the TTR of a problem that requires operator intervention partially depends on the priority the operator places on diagnosing and repairing the problem. This priority, in turn, depends on the operator's judgment of the impact of the problem on the service. Some problems are urgent, *e.g.*, a CPU failure in the machine holding the unreplicated database containing the mapping of service user IDs to passwords. In that case repair is likely to be initiated immediately. Other problems, or *even the same problem when it occurs in a different context*, are less urgent, *e.g.*, a CPU failure in one of a hundred redundant front-end nodes is likely to be addressed much more casually than is the database CPU failure. More generally, a problem's priority, as judged by an operator, depends on not only purely technical metrics such as performance degradation, but also on business-oriented metrics such as the importance of the customer(s) affected by the problem or the importance of the part of the service that has experienced the problem (*e.g.*, a service's email system may be considered to be more critical than the system that generates advertisements, or vice-versa).

	Front-end	**Back-end**	**Network**
Online	9.4 (16)	7.3 (5)	7.8 (4)
Content	2.5 (10)	14 (3)	1.2 (2)
Read-Mostly	N/A (0)	0.2 (1)	1.2 (16)

Table 4: Average TTR by part of service, in hours. The number in parentheses is the number of service failures used to compute that average.

[1] As with our definition of "service failure," restoration of the service to its pre-failure QoS is based not on an empirical measurement of system QoS but rather on inference from the system architecture, the component that failed, and the operator log of the repair process.

	Operator node	Operator net	H/W node	H/W net	S/W node	S/W net	Unknown node	Unknown net
Online	8.3 (16)	29 (1)	2.5 (5)	0.5 (1)	4.0 (9)	0.8 (1)	2.0 (1)	N/A (0)
Content	1.2 (8)	N/A (0)	N/A (0)	N/A (0)	0.2 (4)	N/A (0)	N/A (0)	1.2 (2)
Read-Mostly	0.2 (1)	0.1 (3)	N/A (0)	6.0 (2)	N/A (0)	1.0 (4)	N/A (0)	0.1 (6)

Table 5: Average TTR for failures by component and type of cause, in hours. The component is described as node or network, and failure cause is described as operator error, hardware, software, unknown, or environment. The number in parentheses is the number of service failures used to compute that average. We have excluded the "overload" category because of the very small number of failures due to that cause.

4. Techniques for mitigating failures

Given that user-visible failures are inevitable despite these services' attempts to prevent them, how could the service failures that we observed have been avoided, or their impact reduced? To answer this question, we analyzed 40 service failures from *Online*, asking whether any of a number of techniques that have been suggested for improving availability could potentially

- prevent the original component design flaw (fault)
- prevent a component fault from turning into a component failure
- reduce the severity of degradation in user-perceived QoS due to a component failure (*i.e.,* reduce the degree to which a service failure is observed)
- reduce the Time to Detection (TTD): time from component failure to detection of the failure
- reduce the Time to Repair (TTR): time from component failure detection to component repair. (This interval corresponds to the time during which system QoS is degraded.)

Figure 6 shows how these categories can be viewed as a state machine or timeline, with component fault leading to component failure, possibly causing a user-visible service failure; the component failure is eventually detected, diagnosed, and repaired, returning the system to its failure-free QoS.

The techniques we investigate for their potential effectiveness were

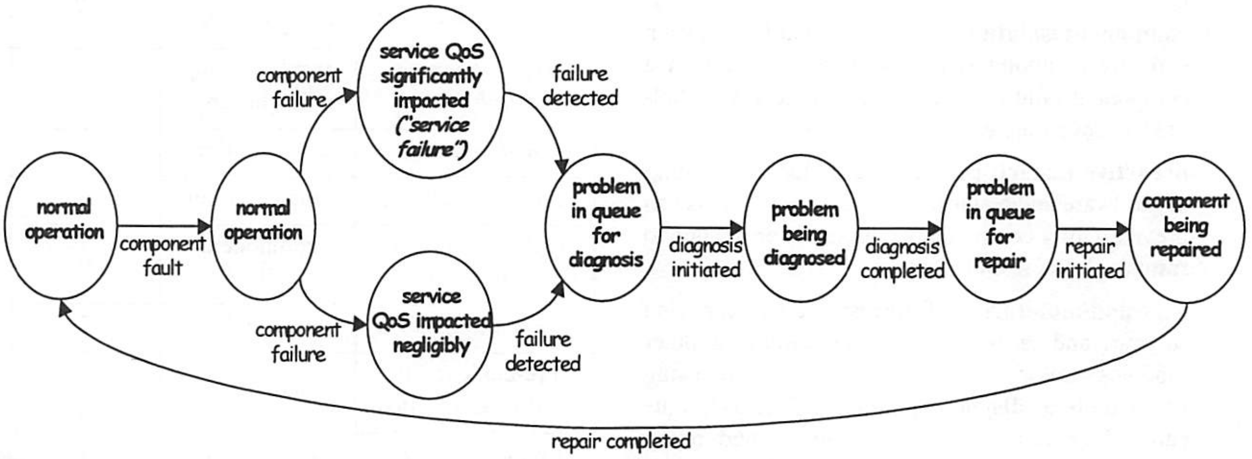


Figure 6: Timeline of a failure. The system starts out in *normal operation*. A *component fault*, such as a software bug, an alpha particle flipping a memory bit, or an operator misunderstanding the configuration of the system he or she is about to modify, may or may not eventually lead the affected component to fail. A *component failure* may or may not significantly impact the service's QoS. In the case of a simple component failure, such as an operating system bug leading to a kernel panic, the component failure may be automatically *detected* and *diagnosed* (*e.g.*, the operating system notices an attempt to twice free a block of kernel memory), and the *repair* (initiating a reboot) will be automatically initiated. A more complex component failure may require operator intervention for detection, diagnosis, and/or repair. In either case, the system eventually returns to normal operation. In our study, we use TTR to denote the time between "failure detected" and "repair completed."

- **correctness testing:** testing the system and its components for correct behavior before deployment or in production. Pre-deployment testing prevents component faults in the deployed system, and online testing detects faulty components before they fail during normal operation. Online testing will catch those failures that are unlikely to be created in a test situation, for example those that are scale- or configuration-dependent.
- **redundancy:** replicating data, computational functionality, and/or networking functionality [5]. Using sufficient redundancy often prevents component failures from turning into service failures.
- **fault injection and load testing:** testing error-handling code and system response to overload by artificially introducing failure and overload, before deployment or in the production system [18]. Pre-deployment, this aims to prevent components that are faulty in their error-handling or load-handling capabilities from being deployed; online, this detects components that are faulty in their error-handling or load-handling capabilities before they fail to properly handle anticipated faults and loads.
- **configuration checking:** using tools to check that low-level (*e.g.*, per-component) configuration files meet constraints expressed in terms of the desired high-level service behavior [13]. Such tools could prevent faulty configurations in deployed systems.
- **component isolation:** increasing isolation between software components [5]. Isolation can prevent a component failure from turning into a service failure by preventing cascading failures.
- **proactive restart:** periodic prophylactic rebooting of hardware and restarting of software [7]. This can prevent faulty components with latent errors due to resource leaks from failing.
- **exposing/monitoring failures:** better exposing software and hardware component failures to other modules and/or to a monitoring system, or using better tools to diagnose problems. This technique can reduce time to detect, diagnose, and repair component failures, and it is especially important in systems with built-in redundancy that masks component failures.

Of course, in implementing online testing, online fault injection, and proactive restart, care must be taken to avoid interfering with the operational system. A service's existing partitioning and redundancy may be exploited to prevent these operations from interfering with the service delivered to end-users, or additional isolation might be necessary.

Table 6 shows the number of problems from *Online*'s problem tracking database for which use, or more use, of each technique could potentially have prevented the problem that directly caused the system to enter the corresponding failure state. A given technique generally addresses only one or a few system failure states; we have listed only those failure states we consider feasibly addressed by the corresponding technique. Because our analysis is made in retrospect, we tried to be particularly careful to assume a *reasonable* application of each technique. For example, using a trace of past failed and successful user requests as input to an online regression testing mechanism would be considered reasonable after a software change, whereas creating a bizarre combination of inputs that seemingly incomprehensibly triggers a failure would not.

Note that if a technique prevents a problem from causing the system to enter some failure state, it also necessarily prevents the problem from causing the system to enter a subsequent failure state. For example,

Technique	**System state or transition avoided/ mitigated**	**instances potentially avoided/ mitigated**
Online correctness testing	component failure	26
Expose/monitor failures	component being repaired	12
Expose/monitor failures	problem being diagnosed	11
Redundancy	service failure	9
Config. checking	component fault	9
Online fault/load injection	component failure	6
Component isolation	service failure	5
Pre-deployment fault/load injection	component fault	3
Proactive restart	component fail	3
Pre-deployment correctness testing	component fault	2

Table 6: Potential benefit from using in *Online* various proposed techniques for avoiding or mitigating failures. 40 service failures were examined, taken from the same time period as those analyzed in Section 3.3. Those techniques that *Online* is already using are indicated in italics; in those cases we evaluate the benefit from using the technique more extensively.

preventing a component fault prevents the fault from turning into a failure, a degradation in QoS, and a need to detect, diagnose, and repair the failure. Note that techniques that reduce time to detect, diagnose, or repair component failure reduce overall service loss experienced (*i.e.,* the amount of QoS lost during the failure multiplied by the length of the failure).

From Table 6 we observe that online testing would have helped the most, mitigating 26 service failures. The second most helpful technique, more thoroughly exposing and monitoring for software and hardware failures, would have decreased TTR and/or TTD in more than 10 instances. Simply increasing redundancy would have mitigated 9 failures. Automatic sanity checking of configuration files, and online fault and load injection, also appear to offer significant potential benefit. Note that of the techniques, *Online* already uses some redundancy, monitoring, isolation, proactive restart, and pre-deployment and online testing, so Table 6 underestimates the effectiveness of adding those techniques to a system that does not already use them.

Naturally, all of the failure mitigation techniques described in this section have not only benefits, but also costs. These costs may be financial or technical. Technical costs may come in the form of a performance degradation (*e.g.,* by increasing service response time or reducing throughput) or reduced reliability (if the complexity of the technique means bugs are likely in the technique's implementation). Table 7 analyzes the proposed failure mitigation techniques with respect to their costs. With this cost tradeoff in mind, we observe that the techniques of adding additional redundancy and better exposing and monitoring for failures offer the most significant "bang for the buck," in the sense that they help mitigate a relatively large number of failure scenarios while incurring relatively low cost.

Clearly, better online correctness testing could have mitigated a large number of system failures in *Online* by exposing latent component faults before they turned into failures. The kind of online testing that would have helped is fairly high-level self-tests that require application semantic information (*e.g.,* posting a news article and checking to see that it showed up in the newsgroup, or sending email and checking to see that it is received correctly and in a timely fashion). Unfortunately these kinds of tests are hard to write and need to be changed every time the service functionality or interface changes. But, qualitatively we can say that this kind of testing would have helped the other services we examined as well, so it seems a useful technique.

Online fault injection and load testing would likewise have helped *Online* and other services. This observation goes hand-in-hand with the need for better exposing failures and monitoring for those failures--online fault injection and load testing are ways to ensure that component failure monitoring mechanisms are correct and sufficient. Choosing a set of representative faults and error conditions, instrumenting code to inject them, and then monitoring the response, requires potentially even more work than does online correctness testing. Moreover, online fault injection and load testing require a performance- and reliability-isolated subset of the production service to be used, because of the threat they pose to the performance and reliability of the production system. But we found that, despite the best intentions, offline test clusters tend to be set up slightly differently than the production cluster, so the online approach appears to offer more potential benefit than does the offline version.

5. Failure case studies

In this section we examine in detail a few of the more instructive service failures from *Online*, and one failure from *Content* related to a service provided to the operations staff (as opposed to end-users).

Our first case study illustrates an operator error affecting front-end machines. In that problem, an operator at *Online* accidentally brought down half of the front-end servers for one service group (partition of users) using the same administrative shutdown com-

Technique	Implementation cost	Potential reliability cost	Performance impact
Online-correct	medium to high	low to moderate	low to moderate
Expose/ monitor	medium	low (false alarms)	low
Redundancy	low	low	very low
Online-fault/load	high	high	moderate to high
Config	medium	zero	zero
Isolation	moderate	low	moderate
Pre-fault/ load	high	zero	zero
Restart	low	low	low
Pre-correct	medium to high	zero	zero

Table 7: Costs of implementing failure mitigation techniques described in this section.

mand issued separately to three of the six servers. Only one technique, redundancy, could have mitigated this failure: because the service had neither a remote console nor remote power supply control to those servers, an operator had to physically travel to the colocation site and reboot the machines, leading to 37 minutes during which users in the affected service group experienced 50% performance degradation when using "stateful" services. Remote console and remote power supply control are a redundant control path, and hence a form of redundancy. The lesson to be learned here is that improving the redundancy of a service sometimes cannot be accomplished by further replicating or partitioning existing data or service code. Sometimes redundancy must come in the form of orthogonal redundancy, such as a backup control path.

A second interesting case study is a software error affecting the service front-end; it provides a good example of a cascading failure. In that problem, a software upgrade to the front-end daemon that handles username and alias lookups for email delivery incorrectly changed the format of the string used by that daemon to query the back-end database that stores usernames and aliases. The daemon continually retried all lookups because those looks were failing, eventually overloading the back-end database, and thus bringing down all services that used the database. The email servers became overloaded because they could not perform the necessary username/alias lookups. The problem was finally fixed by rolling back the software upgrade and rebooting the database and front-end nodes, thus relieving the database overload problem and preventing it from recurring.

Online testing could have caught this problem, but pre-deployment component testing did not, because the failure scenario was dependent on the interaction between the new software module and the unchanged back-end database. Throttling back username/alias lookups when they started failing repeatedly during a short period of time would also have mitigated this failure. Such a use of isolation would have prevented the database from becoming overloaded and hence unusable for providing services other than username/alias lookups.

A third interesting case study is an operator error affecting front-end machines. In this situation, users noticed that their news postings were sometimes not showing up on the service's newsgroups. News postings to local moderated newsgroups are received from users by the front-end news daemon, converted to email, and then sent to a special email server. Delivery of the email on that server triggers execution of a script that verifies the validity of the user posting the message. If the sender is not a valid *Online* user, or the verification otherwise fails, the server silently drops the message. A service operator at some point had configured that email server not to run the daemon that looks up usernames and aliases, so the server was silently dropping all news-postings-converted-into-email-messages that it was receiving. The operator accidentally configured that email server not to run the lookup daemon because he or she did not realize that proper operation of that mail server depended on its running that daemon.

The lessons to be learned here are that software should never silently drop messages or other data in response to an error condition, and perhaps more importantly that operators need to understand the high-level dependencies and interactions among the software modules that comprise a service. Online testing would have detected this problem, while better exposing failures, and improved techniques for diagnosing failures, would have decreased the time needed to detect and localize this problem. Online regression testing should take place not only after changes to software components, but also after changes to system configuration.

A fourth failure we studied arose from a problem at the interface between *Online* and an external service. *Online* uses an external provider for one of its services. That external provider made a configuration change to its service to restrict the IP addresses from which users could connect. In the process, they accidentally blocked clients of *Online*. This problem was difficult to diagnose because of a lack of thorough error reporting in *Online*'s software, and poor communication between *Online* and the external service during problem diagnosis and when the external service made the change. Online testing of the security change would have detected this problem.

Problems at the interface between providers is likely to become increasingly common as composed network services become more common. Indeed, techniques that could have prevented several failures described in this section--orthogonal redundancy, isolation, and understanding the high-level dependencies among software modules--are likely to become more difficult, and yet essential to reliability, in a world of planetary-scale ecologies of networked services.

As we have mentioned, we did not collect statistics on problem reports pertaining to systems whose failure could not directly affect the end-user experience. In particular, we did not consider problem reports pertaining to hardware and software used to support system administration and operational activities. But one incident merits special mention as it provides an excellent example of multiple related, but non-cascading, component failures contributing to a single failure. Ironically, this problem led to the destruction of *Online*'s entire problem tracking database while we were conducting our research.

Content's problem tracking database was stored in a commercial database. The data was supposed to be backed up regularly to tape. Additionally, the data was remotely mirrored each night to a second machine, just as the service data itself was remotely mirrored each night to a backup datacenter. Unfortunately, the database backup program had not been running for a year and a half because of a configuration problem related to how the database host connects to the host with the tape drive. This was considered a low-priority issue because the remote mirroring still ensured the existence of one backup copy of the data. One night, the disk holding the primary copy of the problem tracking database failed, leaving the backup copy as the only copy of the database. In a most unfortunate coincidence, an operator reimaged the host holding the backup (and now only) copy of the database later that evening, before it was realized that the primary copy of the problem tracking database had been destroyed, and that the re-imaging would therefore destroy the last remaining copy.

We can learn several lessons from this failure. First, lightning *does* sometimes strike twice--completely unrelated component failures can happen simultaneously, leading a system with one level of redundancy to fail. Second, categorizing failures, particularly operator error, can be tricky. For example, was reimaging the backup machine after the primary had failed really an operator error, if the operator was unaware that the primary had failed? Was intentionally leaving the tape backup broken an operator error? (We do consider both to be operator errors, but arguably understandable ones.) Third, it is vital that operators understand the current configuration and state of the system and the architectural dependencies and relationships among components. Many systems are designed to mask failures--but this can prevent operators from knowing when a system's margin of safety has been reduced. The correct operator behavior in the previous problem was to replicate the backup copy of the database before reimaging the machine holding it. But, in order to know to do this, the operator needed to know that the primary copy of the problem tracking database had been destroyed, and that the machine he or she was about to reimage held the backup copy of that database. Understanding *how* a system will be affected by a change is particularly important before embarking on destructive operations that are impossible to undo, such as reimaging a machine.

6. Discussion

In this section we describe three areas currently receiving little research attention that we believe could help substantially to improve the availability of Internet services: better operator tools and systems; creation of an industry-wide failure repository; and adoption of standardized service-level benchmarks. We also comment on the representativeness of the data we have presented.

6.1. Operators as first-class users

Despite the huge contribution of operator error to service failure, operator error is almost completely overlooked in designing high-dependability systems and the tools used to monitor and control them. This oversight is particularly problematic because as our data shows, operator error is the most difficult component failure to mask through traditional techniques. Industry has paid a great deal of attention to the end-user experience, but has neglected tools and systems used by operators for configuration, monitoring, diagnosis, and repair.

As previously mentioned, the majority of operator errors leading to service failure were misconfigurations. Several techniques could improve this situation. One is improved operator interfaces. This does not mean a simple GUI wrapper around existing per-component command-line configuration mechanisms--we need fundamental advances in tools to help operators understand existing system configuration and component dependencies, and how their changes to one component's configuration will affect the service as a whole. Tools to help visualize existing system configuration and dependencies would have averted some operator errors (configuration-related and otherwise) by ensuring that an operator's mental model of the existing system configuration matched the true configuration.

Another approach is to build tools that do for configuration files what *lint* [8] does for C programs: to check configuration files against known constraints. Such tools can be built incrementally, with support for additional types of configuration files and constraints added over time. This idea can be extended in two ways. First, support can be added for user-defined constraints, taking the form of a high-level specification of desired system configuration and behavior, much as [3] can be viewed as a user-extensible version of *lint*. Second, a high-level specification can be used to automatically generate per-component configuration files. A high-level specification of operator intent is a form of semantic redundancy, a technique that is useful for catching errors in other contexts (types in programming languages, data structure invariants, and so on). Unfortunately there are no widely used generic tools to allow an operator to specify in a high-level way the desired service architecture and behavior, such that the specification could be checked against the existing configuration, or per-component configurations could be generated. Thus the very wide configuration interface remains error-prone.

An overarching difficulty related to diagnosing and repairing problems relates to collaborative problem solving. Internet services require coordinated activity by multiple administrative entities, and multiple individuals within each of those organizations, for diagnosing and solving some problems. These entities include the operations staff of the service, the service's software developers, the operators of the collocation facilities that the service uses, the network providers between the service and its collocation facilities, and sometimes the customers. Today, this coordination is handled almost entirely manually, via telephone calls to contacts at the various points. The process could be greatly improved by sharing a unified problem tracking/bug database among all of these entities and deploying collaboration tools for cooperative work.

Besides allowing for collaboration, one of the most useful properties of a problem tracking database is that it gives operators a *history* of all the actions that were performed on the system and an indication of why the actions were performed. Unfortunately this history is human-generated in the form of operator annotations to a problem report as they walk through the steps of diagnosing and repairing a problem. A tool that, in a structured way, expresses the history of a system--including configuration and system state before and after each change, who or what made the change, why they made the change, and exactly what changes they made--would help operators understand how a problem evolved, thereby aiding diagnosis and repair.

Tools for determining the root cause of problems across administrative domains, *e.g., traceroute,* are rudimentary, and these tools generally cannot distinguish between certain types of problems, such as end-host failures and network problems on the network segment where a node is located. Moreover, tools for fixing a problem once its source is located are controlled by the administrative entity that owns the broken hardware or software, not by the site that determines that the problem exists. These difficulties lead to increased diagnosis and repair times. The need for tools and techniques for problem diagnosis and repair that work effectively across administrative boundaries, and that correlate system observations from multiple network vantage points, is likely to become even greater in the age of composed network services built on top of emerging platforms such as Microsoft's .NET and Sun's SunONE.

Finally, the systems and tools operators use to administer services are not just primitive and difficult to use, they are also brittle. Although we did not collect detailed statistics about failures in systems used for service operations, we observed many reports of failures of such components. At *Content*, more than 15% of the reports in the problem tracking database related to administrative machines and services. A qualitative analysis of these problems across the services reveals that organizations do not build the operational side of their services with as much redundancy or pre-deployment quality control as they do the parts of the service used by end-users. The philosophy seems to be that operators can work around problems with administrative tools and machines if something goes wrong, while end-users are powerless in the face of service problems. Unfortunately the result of this philosophy is increased time to detect, diagnose, and repair problems due to fragile administrative systems.

6.2. A worldwide failure data repository

Although analyzing failure data seems at first straightforward, our initial expectation turned out to be far from the truth. Because the Internet services we studied recorded component failures in a database, we expected to be able to simply write a few database queries to collect the quantitative data we have presented in this paper. Unfortunately, we found that operators frequently filled out the database forms incorrectly--for example, fields such as problem starting time, problem ending time, root cause, and whether a problem was customer impacting (*i.e.,* a "service failure" as opposed to just a "component failure"), often contradicted the timestamped operator narrative of events that accompanied the problem reports. The data presented in this paper was therefore gathered by reading the operator narrative for each problem report, rather than accepting the pre-analyzed database data on blind faith. Likewise, insufficiently detailed problem reports sometimes led to difficulty in determining the actual root cause of a problem or its time to repair. Finally, the lack of historical end-user-perceived service QoS measurements prevented us from rigorously defining a "service failure" or even calculating end-user-perceived service availability during the time period corresponding to the problem reports we examined.

We believe that Internet services should follow the lead of other fields, such as aviation, in collecting and publishing detailed industry-wide failure-cause data in a standardized format. Only by knowing why real systems fail, and what impact those failures have, can researchers and practitioners know where to target their efforts. Many services, such as the ones we studied, already use databases to store their problem tracking information. It should be possible to establish a standard schema, perhaps extensible, for Internet services, network providers, colocation facilities, and related entities, to use for recording problems, their impact, and their resolutions. We have proposed one possible failure cause and location taxonomy for such a schema in this paper. A standard schema would benefit not only researchers, but also these services themselves, as establishing and sharing such databases would help to address the coordination problem described in Section 6.1. Finally, we note that services that release such data are likely to want the publicly-available version of the database to be anonymized; automating the necessary anonymization is nontrivial, and is a research question unto itself.

6.3. Performability and recovery benchmarks

In addition to focusing dependability research on real-world problem spots, the failure data we have collected can be used to create a fault model (or, to use our terminology, component failure model) for *service-level performability benchmarks*. Recent benchmarking efforts have focused on component-level dependability by observing single-node application or OS response to misbehaving disks, system calls, and the like. But because we found a significant contribution to service failure of human error (particularly multi-node configuration problems) and network (including WAN) problems, we suggest a more holisitc approach. In service-level performability benchmarks, a small-scale replica (or a physically or virtually isolated partition) of a service is created, and QoS for a representative service workload mix is measured while representative component failures (*e.g.*, those described in this paper) are injected. To simplify this process, one might measure the QoS impact of individual component failures or multiple simultaneous failures, and then weight the degraded QoS response to these events by either the relative frequency with which the different classes of component failure occur in the service being benchmarked, or using the proportions we found in our survey. Important metrics to measure in addition to QoS impact of injected failures, are time to detect, diagnose, and repair the component failure(s), be it automatically or by a human. As suggested in [2], the workload should include standard service administrative tasks. A recent step towards this type of benchmark is described in [16].

6.4. Representativeness

While our data was taken from just three services, we feel that it is representative of large-scale Internet services that use custom-written software to provide their service. Most of the "giant scale" services we informally surveyed use custom-written software, at least for the front-end, for scalability and performance reasons. Based on this information, we feel that our results do apply to many Internet services.

On the other hand, our data is somewhat skewed by the fact that two of our three sites (*Content* and *ReadMostly*) are what we would call "content-intensive," meaning that the time spent transferring data from the back-end media to the front-end node is large compared to the amount of time the front-end node spends processing the request and response. Sites that are less "content intensive" are less likely to use custom-written back-end software (as was the case for *Online*). Additionally, at all three services we studied, user requests do not require transactional semantics. Sites that require transactional semantics (*e.g.*, e-commerce sites) are more likely to use database back-ends rather than custom-written back-end software. In theory both of these factors should tend to decrease the failure rate of back-end software, and indeed *Online*, the one site that used off-the-shelf back-end software, was also the site with the lowest fraction of back-end service failures.

7. Related work

Our work adds to a small body of existing studies of, and suggestions for, Internet service architectures [1] [14]. We are not aware of any studies of failure causes of such services.

A number of studies have been published on the causes of failure in various types of computer systems that are not commonly used for running Internet sites, and in operational environments unlike those of Internet services. Gray is responsible for the most widely cited studies of computer system failure data [5] [6]. In 1986 he found that operator error was the largest single cause of failure in deployed Tandem systems, accounting for 42% of failures, with software the runner-up at 25%. We found strikingly similar percentages at *Online* and *Content*. In 1989, however, Gray found that software had become the major source of outages (55%), swamping the second largest contributor, system operations (15%). We note that Gray attributed a failure to the *last* component in a failure chain rather than the root cause, making his statistics not directly comparable to ours, although this fact only matters for cascading failures, of which we found few. Though we did not quantitatively analyze the length of failure chains, Gray found longer chains than we did: 20% of failure chains were of length three or more in Gray's study, with only 20% of length one, whereas we found that almost all were of length one or two.

Kuhn examined two years of data on failures in the Public Switched Telephone Network as reported to the

FCC by telephone companies [10]. He concluded that human error was responsible for more than 50% of failure incidents, and about 30% of customer minutes, *i.e.*, number of customers impacted multiplied by number of minutes the failure lasted. Enriquez extended this study by examining a year's worth of more recent data and by examining the "blocked calls" metric collected on the outage reports [4]. She came to a similar conclusion--human error was responsible for more than 50% of outages, customer-minutes, and blocked calls.

Several studies have examined failures in networks of workstations. Thakur examined failures in a network of 69 SunOS workstations but divided problem root cause coarsely into network, non-disk machine problems, and disk-related machine problems [21]. Kalyanakrishnam studied six months of event logs from a LAN of Windows NT workstations used for mail delivery, to determine the causes of machines rebooting [9]. He found that most problems were software-related, and that average downtime was two hours. In a closely related study, Xu examined a network of Windows NT workstations used for 24x7 enterprise infrastructure services, again by studying the Windows NT event log entries related to system reboots [22]. Unlike the Thakur or Kalyanakrishnam studies, this one allowed operators to annotate the event log to indicate the reason for reboot; thus the authors were able to draw conclusions about the contribution of operator failure to system outages. They found that planned maintenance and software installation and configuration caused the largest number of outages, and that system software and planned maintenance caused the largest amount of total downtime. They were unable to classify a large percentage of the problems (58%). We note that they counted reboots after installation or patching of software as a "failure." Their software installation/configuration category therefore is not comparable to our operator failure category, despite its being named somewhat similarly.

A number of researchers have examined the causes of failures in enterprise-class server environments. Sullivan and Chillarege examined software defects in MVS, DB2, and IMS [19]. Tang and Iyer conducted a similar study for the VAX/VMS operating system in two VAXclusters [20]. Lee and Iyer categorized software faults in the Tandem GUARDIAN operating system [12]. Murphy and Gent examined causes of system crashes in VAX systems between 1985 and 1993; in 1993 they found that hardware caused about 10% of failures, software about 20%, and system management a bit over 50% [15].

Our study is similar in spirit to two studies of network failure causes: Mahajan examined the causes of BGP misconfigurations [13], and Labovitz conducted an earlier study of the general causes of failure in IP backbones [11].

8. Conclusion

From a study of more than 500 component failures and dozens of user-visible failures in three large-scale Internet services, we observe that (1) operator error is the leading cause of failure in two of the three services studied, (2) operator error is the largest contributor to time to repair in two of the three services, (3) configuration errors are the largest category of operator errors, (4) failures in custom-written front-end software are significant, and (5) more extensive online testing and more thoroughly exposing and detecting component failures would reduce failure rates in at least one service.

While 100% availability is almost certainly unattainable, our observations suggest that Internet service availability could be significantly enhanced with proactive online testing; thoroughly exposing and monitoring for component failures; sanity checking configuration files; logging operator actions; and improving operator tools for distributed, collaborative diagnosis and problem tracking. To the extent that tools for these tasks already exist, they are generally *ad hoc* and are retrofitted onto existing systems. We believe it is important that service software be built from the ground up with concern for testing, monitoring, diagnosis, and maintainability in mind--in essence, treating operators as first-class users. To accomplish this, APIs for emerging services should allow for easy online testing, fault injection, automatic propagation of errors to code modules and/or operators that can handle them, and distributed data-flow tracing to help detect, diagnose, and debug performance and reliability failures. Finally, we believe that research into system reliability would benefit greatly from an industry-wide, publicly-accessible failure database, and from service-level performability and recovery benchmarks that can objectively evaluate designs for improved system availability and maintainability.

Acknowledgements

We extend our sincere gratitude to the operations staff and management at the Internet services who provided us data and answered our questions about the data. This work was only possible because of their help. We regret that we cannot thank these services and individuals by name due to our confidentiality agreements with them. We also thank our shepherd Hank Levy and the anonymous reviewers for their very helpful feedback on earlier versions of this paper, and the members of the Berkeley/Stanford ROC group for contributing to the ideas presented in this paper.

This work was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense, contract no. DABT63-96-C-0056, the National Science Foundation, grant no. CCR-0085899, NSF infrastructure grant no. EIA-9802069, Alocity, Microsoft, and the California State MICRO Program. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing,* vol. 5, no. 4, 2001.

[2] A. Brown and D. A. Patterson. To Err is Human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY* (EASY '01), 2001.

[3] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[4] P. Enriquez, A. Brown, and D. A. Patterson. Lessons from the PSTN for dependable computing. *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

[5] J. Gray. A census of Tandem system availability between 1985 and 1990. Tandem Computers Technical Report 90.1, 1990.

[6] J. Gray. Why do computers stop and what can be done about it? *Symposium on Reliability in Distributed Software and Database Systems,* 1986.

[7] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, models, and applications. *25th symposium on fault-tolerant computing,* 1995.

[8] S. C. Johnson. Lint, a C program checker. Bell Laboratories computer science technical report, 1978.

[9] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a LAN of Windows NT based computers. *18th IEEE Symposium on Reliable Distributed Systems,* 1999.

[10] D. R. Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer* 30(4), 1997.

[11] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of Internet stability and backbone failures. *Fault-Tolerant Computing Symposium (FTCS)*, 1999.

[12] I. Lee and R. Iyer. Software dependability in the Tandem GUARDIAN system. *IEEE Transactions on Software Engineering,* 21(5), 1995.

[13] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. *SIGCOMM '02*, 2002.

[14] Microsoft TechNet. Building scalable services. http://www.microsoft.com/technet/treeview/default.asp?url=/TechNet/itsolutions/ecommerce/deploy/projplan/bss1.asp, 2001.

[15] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International,* vol 11, 1995.

[16] K. Nagaraja, X. Li, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault injection and modeling to evaluate the performability of cluster-based services. *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[17] D. Oppenheimer and D. A. Patterson. Architecture, operation, and dependability of large-scale Internet services: three case studies. *IEEE Internet Computing,* September/October, 2002.

[18] D. A. Patterson., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, 2002.

[19] M. S. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing,* 1992.

[20] D. Tang and R. Iyer. Analysis of the VAX/VMS error logs in multicomputer environments--a case study of software dependability. *Proceedings of the Third International Symposium on Software Reliability Engineering,* 1992.

[21] A. Thakur and R. Iyer. Analyze-NOW--an environment for collection adn analysis of failures in a network of workstations. *IEEE Transactions on Reliability,* R46(4), 1996.

[22] J. Xu, Z. Kalbarczyk, and R. Iyer. Networked Windows NT system field failure data analysis. *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing,* 1999.

Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services*

Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854
{knagaraj, xili, ricardob, rmartin, tdnguyen}@cs.rutgers.edu

Abstract. *We propose a two-phase methodology for quantifying the performability (performance and availability) of cluster-based Internet services. In the first phase, evaluators use a fault-injection infrastructure to measure the impact of faults on the server's performance. In the second phase, evaluators use an analytical model to combine an expected fault load with measurements from the first phase to assess the server's performability. Using this model, evaluators can study the server's sensitivity to different design decisions, fault rates, and environmental factors. To demonstrate our methodology, we study the performability of 4 versions of the PRESS Web server against 5 classes of faults, quantifying the effects of different design decisions on performance and availability. Finally, to further show the utility of our model, we also quantify the impact of two hypothetical changes, reduced human operator response time and the use of RAIDs.*

1 Introduction

Popular Internet services frequently rely on large clusters of commodity computers as their supporting infrastructure [5]. These services must exhibit several characteristics, including high performance, scalability, and availability. The performance and scalability of cluster-based servers have been studied extensively in the literature, e.g., [2, 5, 7]. In contrast, understanding designs for availability, behavior during component faults, and the relationship between performance and availability of these servers have received much less attention.

Although today's service designers are not oblivious to the importance of high availability, e.g., [5, 12, 28], the design and evaluation of availability is often based on the practitioner's experience and intuition rather than a quantitative methodology.

In this paper, we advance the state-of-the-art by developing a 2-phased methodology that combines fault-injection and analytical modeling to study and quantify the performability – a metric combining performance *and* availability – of cluster-based servers. In the first phase of our methodology, the server is benchmarked for performance and availability both in the presence and absence of faults. To support systematic fault-injection, we introduce Mendosus, a fault-injection and network emulation infrastructure designed specifically to study cluster-based servers. While evaluators using our methodology are free to use any fault-injection framework, Mendosus provides significant flexibility in emulating different LAN configurations and is able to inject a wide variety of faults, including link, switch, disk, node, and process faults.

The second phase of our methodology uses an analytical model to combine an expected fault model [23, 32], measurements from the first phase, and parameters of the surrounding environment to predict performability. Designers can use this model to study the potential impact of different design decisions on the server's behavior. We introduce a single performability measure to enable designers to easily characterize and compare servers.

To show the practicality of our methodology, we use it to study the performability of PRESS, a cluster-based Web server [7]. A significant benefit of analyzing PRESS is that, over time, the designers of PRESS have accumulated different versions with varying levels of performance. Using our methodology, we can quantify the impact of changes from one version to another on availability, and therefore, performability, producing a more complete picture than just the previous data on performance. For example, a PRESS version using TCP for intra-cluster communication achieves a higher overall performability score even though it does not perform as well as a version using VIA. We also show how our model can be used to predict the impact of design or environmental changes; in particular, we use our model to study PRESS's sensitivity to operator coverage and using RAIDs instead of independent SCSI disks.

We make the following contributions:

- We propose a methodology that combines fault injection, experimentation, and modeling to quantify a server's availability as well as its performance.
- We demonstrate the power of our methodology by

*This work was supported in part by NSF grants EIA-0103722 and EIA-9986046

using it to evaluate four different versions of a sophisticated cluster-based server. We also quantitatively evaluate design and environmental tradeoffs on the server's performability.

- We use results from our study to derive several guidelines on how to design highly available cluster-based servers.

The remainder of the paper is organized as follows. The next section describes our methodology and performability metric. Section 3 describes our fault-injection infrastructure. Section 4 describes the basic architecture of the PRESS server and its different versions. Section 5 presents the results of our fault-injection experiments into the live server. Section 6 describes the results of our analytical modeling of PRESS. We discuss the lessons we learned in Section 7. Section 8 describes the related work. Finally, in Section 9 we draw our most important conclusions.

2 Methodology and Metric

Our methodology for evaluating servers' performability is comprised of two phases. In the first phase, the evaluator defines the set of all possible faults, then injects them (and the subsequent recovery) one at a time into a running system. During the fault and recovery periods, the evaluator must quantify performance and availability as a function of time. We currently equate performance with throughput, *the number of requests successfully served per second*, and define availability as *the percentage of requests served successfully*. In the second phase, the evaluator uses an analytical model to compute the expected average throughput and availability, combining the server's behavior under normal operation, the behavior during component faults, and the rates of fault and repair of each component.

2.1 Phase 1: Measuring Performance Under Single-Fault Fault Loads

There are two tricky issues when injecting faults. First, when measuring the server's performance in the presence of a particular fault, the fault must last long enough to allow all stages in the model of phase 2 to be observed and measured. The one exception to this guideline is that a server may not exhibit all model stages under certain faults. In these cases, the evaluator must use his understanding of the server to correctly determine which stages are missing (and later setting the time of the stage in the abstract model to 0). Second, a benchmark must be chosen to drive the server such that the delivered throughput is relatively stable throughout the observation period (except for transient warm up effects). This is necessary to decouple measured performance from the injection time of a fault.

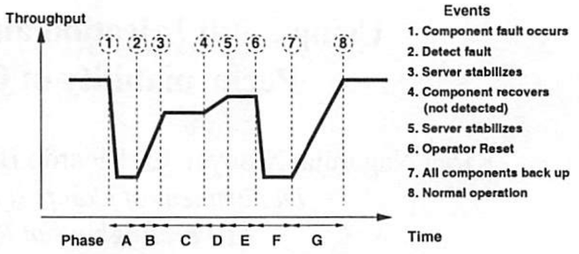


Figure 1: *The 7-stage piece-wise linear model specified by our methodology for evaluating the performability of cluster-based servers.*

2.2 Phase 2: Modeling Performability Under Expected Fault Loads

Our model for describing average performance and availability is built in two parts. The first part of the model describes the system's response to each fault in 7 stages. The second part combines the effects of each fault along with the MTTF (Mean Time To Failure) and MTTR (Mean Time To Recovery) of each component to arrive at an overall average availability and performance.

Per-Fault Seven-Stage Model. Figure 1 illustrates our 7-stage model of service performance in the presence of a fault. Time is shown on the X-axis and throughput is shown on the Y-axis. Stage A models the degraded throughput delivered by the system from the occurrence of the fault to when the system detects the fault. Stage B models the transient throughput delivered as the system reconfigures to account for the fault; the system may take some time to reach a stable performance regime because of warming effects. We model the throughput during this transient period as the average throughput for the period. After the system stabilizes, throughput will likely remain at a degraded level because the faulty component has not yet recovered, been repaired or replaced. Stage C models this degraded performance regime. Stage D models the transient performance after the component recovers. Stage E models the stable performance regime achieved by the service after the component has recovered. Note that in the figure, we show the performance in E as being below that of normal operation; this may occur because the system is unable to reintegrate the recovered component or reintegration does not lead to full recovery. In this case, throughput remains at the degraded level until an operator detects the problem. Stage F represents throughput delivered while the server is reset by the operator. Finally, stage G represents the transient throughput immediately after reset.

For each stage, we need two parameters: (i) the length of time that the system will remain in that stage, and (ii) the average throughput delivered during that stage. The latter is measured in phase 1. The former is either measured, or is a parameter that must be supplied. For example, the time that a service will remain in stage B assuming that the fault last sufficiently long is typically measured; the time a service will remain in stage E is typically a supplied parameter.

Sometimes stages may not be present or may be cut short. For example, if there are no warming effects, then stages B, D, and G would not exist. In practice, we set the length of time the system is in such a state to zero. If the assumed MTTR of a component is less than the measured time for stages A and B, then we assume that B is cut short when the component recovers. The evaluator must analyze the measurements gathered in phase 1, the assumed parameters of the fault load, and the environment carefully to correctly parameterize the model.

Modeling Overall Availability and Performance. Having defined the server's response to each fault, we now must combine all these effects into an average performance and average availability metric. To simplify the analysis, we assume that faults of different components are not correlated, fault arrivals are exponentially distributed, and faults queue at the system so that only a single fault is in effect at any point in time. These assumptions allow us to add together the various fractions of time spent in degraded modes. If T_n is the server throughput under normal operation, c is the faulty component, T_c^s is the throughput of each stage s when fault c occurs, and D_c^s is the duration of each stage, our model leads to the following equations for average throughput (AT) and average availability (AA):

$$AT = (1 - \sum_c W_c)T_n + \sum_c \sum_{s=A}^{G} (\frac{D_c^s}{MTTF_c} T_c^s)$$

$$AA = \frac{AT}{T_n}$$

where $W_c = (\sum_{s=A}^{G} D_c^s)/MTTF_c$. In plain English, W_c is the expected fraction of the time during which the system operates in the presence of fault c. Thus, the $(1 - \sum_c W_c)T_n$ factor above computes the expected throughput when the system is free of any fault, whereas the $\sum_{s=A}^{G}(\frac{D_c^s}{MTTF_c}T_c^s)$ factor computes the expected average throughput when the system is operating with a single fault of type c. Note that T_n represents the offered load assuming that the server is not saturated under normal operation, so AT/T_n computes the expected fraction of offered requests that are successfully served by the system.

It is interesting to consider why the denominator of W_c is just $MTTF_c$ instead of $MTTF_c + MTTR_c$. The equation for W_c is correct as it is because the assumptions listed above imply that when a fault occurs and is on-going, any other fault could arrive and queue at the system, including a fault to the same component. The impact on our model is that we compute the fraction of downtime as $\frac{MTTR}{MTTF}$, not as the more typical $\frac{MTTR}{MTTF+MTTR}$. In practice, the numerical impact of this difference is minimal, because $MTTF >> MTTR$.

Limitations. A current limitation of our model is that it does not capture data integrity faults; that is, faults that lead to incorrect data being served to clients. Rather it assumes the only consequence of component faults is degradation in performance or availability. While this model is obviously not general enough to describe all cluster-based servers, we believe that it is representative of a large class of servers, such as front-end servers (including PRESS) and other read-only servers.

Another limitation of our model is that it is based on the measured response to single faults; the model can thus only capture multiple simultaneous faults as a sequence of non-overlapping faults. If we assume that faults are independent, then the introduced error is bounded by the probability of there being two or more jobs in a single multi-class server queue when the fault arrival and repair processes are viewed in a queuing-theoretic framework. Intuition tells us that the probability of seeing multiple simultaneous faults for practical MTTFs and MTTRs should be extremely low. Determining the probability of simultaneous faults exactly is not straightforward, but our initial approximations (assuming the rates in this paper) show that we can expect around 2 multi-fault events per year. On the other hand, there are indications that failures are not always independent [22, 35], as well as anecdotal evidence that baroque, complex failures are not uncommon [14]. These observations imply that the independence assumptions in our model will result in optimistic predictions for the frequency of multi-fault scenarios [33]. Unfortunately, there is no study that quantifies such correlations for cluster-based Internet services. In the future, we may extend our methodology for designers to test their service's sensitivity to sets of potentially correlated faults.

2.3 Performability Metric

Despite much work that studies both performance and availability (e.g., [21, 30]), there is arguably no *single* performability metric for comparing systems. Thus, we propose a combined *performability* metric that allows direct comparison of systems using both performance and availability as input criteria. Our approach is to multiply the average throughput by an availability factor; the challenge, of course, is to derive a factor that properly balances both availability and performance. Because availability is often characterized in terms of "the number of nines" achieved, we believe that a log-scaled ratio of how each server compares to an ideal system is an appropriate availability measure, leading to the following equation for performability:

$$P = T_n \times \frac{\log(A_I)}{\log(AA)}$$

where A_I is an ideal availability, T_n is the throughput under normal operation, AA is the average availability, and P is the performability of the system. A_I must be less than 1 but can otherwise be chosen by the service designers to represent the availability that is desired for the service, e.g., 0.99999.

This metric is an intuitive measure for performability because it scales linearly with both performance and unavailability. Obviously, if performance doubles, our performability metric doubles. On the other hand, if the *unavailability* decreases by a factor of 2, then performability also roughly doubles. The intuition behind this relationship between unavailability (u) and performability is that we can approximate $\log(1 - u)$ as $-u$ when u is small. Further, if the service designers wish to weigh one factor more heavily than the other, their importance can easily be adjusted by multiplying each term by a separate constant weight.

3 Mendosus

Mendosus is a fault injection and network emulation infrastructure designed to support phase 1 of our methodology. Mendosus addresses two specific problems that service designers are faced with today: (1) how to assemble a sufficiently representative test-bed to test a service as it is being built, and (2) how to conveniently introduce faults to study the service's behavior under various fault loads. In this section, we first briefly describe Mendosus's architecture and then discuss the fault models used by the network, disk, and node fault injection modules, which are used extensively in this work, in more detail.

3.1 Architecture

Mendosus is comprised of four software components running on a cluster of PCs physically interconnected by a Giganet VIA network: (1) a central controller, (2) a per-node LAN emulator module, (3) a set of per-node fault-injection kernel modules, and (4) a per-node user-level daemon that serves as the communication conduit between the central controller and the kernel modules.

The central controller is responsible for deciding when and where faults should be injected and for maintaining a consistent view of the entire network. When emulation starts, the controller parses a configuration file that describes the network to be emulated and components' fault profiles. It forwards the network configuration to the daemon running at each node of the cluster. Then, as the emulation progresses, it uses the fault profiles to decide what faults to inject and when they should be injected. It communicates with the per-node daemons as necessary to effect the faults (and subsequent recovery). Note that while there is one central controller per emulated system, it does not limit the scalability of Mendosus: the controller only deals with faults and does not participate in any per message operations.

The per-node emulation module maintains the topology and status of the virtual network to route messages. To emulate routing in Ethernet networks, a spanning tree is computed for the virtual network. Each emulated NIC is presented as an Ethernet device; a node may have multiple emulated NICs. When a packet is handed to the Ethernet driver from the IP layer, the driver invokes the emulation module to determine whether the packet should be forwarded over the real network (and which node it should be forwarded to). The emulation module determines the emulated route that would be taken by the packet. It then queries the network fault-injection module whether the packet should be forwarded. If the answer is yes, the packet is forwarded to the destination over the underlying real network. The emulation module uses multiple point-to-point messages to emulate Ethernet multicast and broadcast. A leaky bucket is used to emulate Ethernet LANs with different speeds.

Finally, the set of fault-injection kernel modules effect the actual faults as directed by the central controller. Currently, we have implemented 3 modules, allowing faults to be injected into the network, SCSI disk subsystems, and individual nodes. The challenge in implementing these subsystems is to accurately understand the set of possible real faults and the fault reporting that percolates from the device through the device drivers, operating systems, and ultimately, up to the application. We describe the fault models we have implemented in more details in the next several sections.

3.2 Network Fault Model

The network fault model includes faults possible for network interface cards, links, hubs, and switches. For each component, a fault can lead to probabilistic packet loss or complete loss of communication. In addition, for switches and hubs, partial failures of one or more ports are possible. All faults are transient although a permanent fault can be injected by specifying a down time that

Fault	Characteristic	OS Masking of Fault
Disk hang	Sticky	Unmasked
Disk offline	Sticky	Unmasked
Power failure	Sticky	Unmasked
Read fault	Sticky	Unmasked in Linux
Write fault	Sticky	Unmasked in Linux
Timeout	Transient	Unmasked
Parity errors	Transient	Masked
Bus busy	Transient	Masked
Queue full	Transient	Masked

Table 1: *SCSI faults that Mendosus can currently inject.*

is greater than the time required to run the fault-injection experiment.

Our fault-injection module is embedded within an emulated Ethernet driver. Recall that the emulated driver also includes our LAN emulator module, which contains all information needed to compute the route that each packet will take. Fault injection for the network subsystem is straightforward when the communication protocol already implements end-to-end fault detection. Faults are effected simply by checking whether all components on the route are up. If any are down, the packet is simply dropped. If any components are in an intermittent faulty state, then the packet is dropped (or sent) according to the specified distribution.

Recall that the central controller is responsible for instructing the network fault injection modules on when, where, and what to inject dynamically. Instructions from the central controller are received by the local daemon and passed to the injection module through the ioctl interface. The fault-injection and emulation modules must work together in that faults may require the emulation module to recompute the routing spanning tree. The central controller is responsible for determining when a set of faults leads to network partition. When this occurs, the controller must choose a root for each partition so that the nodes within the partition can recompute the routing spanning tree.

3.3 SCSI Disk Fault Model

The SCSI subsystem is comprised of the hard disk device, the host adaptor, a SCSI cable connecting the two, and a hierarchy of software drivers. Higher layers in the system try to mask faults at lower levels and only the fatal faults are explicitly passed up the hierarchy. We model faults that are noticeable by the application either by explicitly forcing error codes reported by the operating system, or implicity by extended delays in the completion of disk operations.

We broadly classify possible faults into two categories: *transient* and *sticky (non-transient)*. For transient faults, the disk system recovers after a small finite interval (on order of a few seconds in most cases); sticky faults require human intervention for correction. Examples of transient faults are SCSI timeouts, and recoverable read and write faults, whereas disk hang, external SCSI cable unplugging and power failures (to external SCSI housing) are sticky failures.

The impact of these faults on the system depends on whether the OS can and does attempt to mask the fault from the application through either a retry or some other corrective action. For example, a parity error in the SCSI bus is typically masked by the OS through a retry, whereas the OS does not attempt to mask a disk hang due to a firmware bug because this error likely requires external intervention. Masked faults may introduce tolerable delays, whereas unmasked faults may lead to stalling of execution. However, some unmasked faults, if recognized, can be handled by using alternate resources. This involves implementing smarter, fault-aware systems.

Table 1 shows the faults that we can inject into the SCSI subsystem. The fault injection module is interposed between the adapter-specific low-level driver and the generic mid-level driver. Instructions for injecting faults received from the central controller by the local daemon are communicated to the fault injection module through the proc filesystem. To effect faults, the fault injection module traps the queuing of disk operation requests to the low-level driver and prevents or delays the operation that should be faulty from reaching the low-level driver. In the former case, the module must return an appropriate error message.

The mid-level driver implements an error handler which diagnoses and corrects rectifiable faults reported by the low-level driver, either by retrying the command or by resetting the host, bus, device, or a combination of these. The unmasked read and write faults, caused by bad sectors unremappable by the disk controller are not handled by the upper drivers or the file system in Linux. This causes read and write operations to the bad sector to fail forever. The disk can be taken offline by the new error handler code introduced in 2.2+ Linux kernels when all efforts to rectify an encountered error fail. The disk can also be offline if it has been taken out for maintainence or replacement.

3.4 Node and Process Fault Models

Currently, our node and process fault model is simple. Mendosus can inject three types of node faults: hard reboot, soft reboot, and node freeze. All can be either transient or permanent, depending on the specified fault load. In the application process fault model, Mendosus can inject an application hang or crash. We may consider more subtle node/process faults such as memory corruption in the future.

This fault model is implemented inside the user-level daemon at each node. For our study of PRESS, the server process on each node is started by the daemon. An application hang is injected by having the daemon send a SIGSTOP to the server process. The process can be restarted if the fault is transient by sending a SIGCTN to it. A process crash is injected by killing the process.

Node faults are introduced using an APC power management power strip. Reboot faults are introduced by having the daemon on the failing node contact the APC power strip to power cycle that node. In the case of a soft reboot, the daemon can ask the APC for a delayed power cycle and then run a shutdown script. For a node freeze, the daemon directs a small kernel module to spin endlessly to take over the CPU for some amount of time.

4 The PRESS Server

PRESS is a highly optimized yet portable cluster-based locality-conscious Web server that has been shown to provide good performance in a wide range of scenarios [7, 8]. Like other locality-conscious servers [27, 2, 4], PRESS is based on the observation that serving a request from any memory cache, even a remote cache, is substantially more efficient than serving it from disk, even a local disk. In PRESS, any node of the cluster can receive a client request and becomes the *initial node* for that request. When the request arrives at the initial node, the request is parsed and, based on its content, the node must decide whether to service the request itself or forward the request to another node, the *service node*. The service node retrieves the file from its cache (or disk) and returns it to the initial node. Upon receiving the file from the service node, the initial node sends it to the client.

To intelligently distribute the HTTP requests it receives, each node needs locality and load information about all the other nodes. Locality information takes the form of the names of the files that are currently cached, whereas load information is represented by the number of open connections handled by each node. To disseminate caching information, each node broadcasts its action to all other nodes whenever it replaces or starts caching a file. To disseminate load information, each node piggybacks its current load onto any intra-cluster message.

Communication Architecture. PRESS is comprised of one `main` coordinating thread and a number of helper threads used to ensure that the `main` thread never blocks. The helper threads include a set of `disk` threads used to access files on disk and a pair of `send/receive` threads for intra-cluster communication.

PRESS can use either TCP or VIA for intra-cluster communication. The TCP version basically has the same structure of its VIA counterpart; the main differences are the replacement of the VI end-points by TCP sockets and the elimination of flow control messages, which are implemented transparently to the server by TCP itself.

Reconfiguration. PRESS is often used (as in our experiments) without a front-end device, relying on round-robin DNS for initial request distribution to nodes. Some versions of PRESS have been designed to tolerate node (and application process) crashes, removing the faulty node from the cooperating cluster when the fault is detected and re-integrating the node when it recovers. The detection mechanism when TCP is used for intra-cluster communication employs periodic heartbeat messages. To avoid sending too many messages, we organize the cluster nodes in a directed ring structure. A node only sends heartbeats to the node it points to. If a node does not receive three consecutive heartbeats from its predecessor, it assumes that the predecessor has failed.

Fault detection when VIA is used for intra-cluster communication is simpler. PRESS does not have to send heartbeat messages itself since the communication subsystem promptly breaks a connection on the detection of any fault. Thus, a node assumes that another node has failed if the VIA connection between them is broken. In this implementation, nodes are also organized in a directed ring, but only for recovery purposes.

In both cases, temporary recovery is implemented by simply excluding the failed node from the server. Multiple node faults can occur simultaneously. Every time a fault occurs, the ring structure is modified to reflect the new configuration.

The second and final step in recovery is to re-integrate a recovered node into the cluster. When using TCP, the rejoining node broadcasts its IP address to all other nodes. The currently active node with lowest id responds by informing the rejoining node about the current cluster configuration and its node id. With that information, the rejoining node can reestablish the intra-cluster connections with the other nodes. After each connection is reestablished, the rejoining node is sent the caching information of the respective node. When the intra-cluster communication is done with VIA, the rejoining node simply tries to reestablish its connection with all other nodes. As connections are reestablished, the rejoining node is sent the caching information of the respective nodes.

Versions. Several versions of PRESS have been developed in order to study the performance impact of different communication mechanisms [8]. Table 2 lists the versions of PRESS that we consider in this paper. The base version of PRESS, I-PRESS, is comprised of a number of independent Web servers (based on the same code as PRESS) answering client requests. This is equivalent to simply running multiple copies of Apache, for example.

Version	Main Features
I-PRESS	Independent servers
TCP-PRESS	Cooperative caching servers using TCP for intra-cluster communication
ReTCP-PRESS	Cooperative caching servers using TCP for intra-cluster communication and dynamic reconfiguration
VIA-PRESS	Cooperative caching servers using VIA for intra-cluster communication and dynamic reconfiguration

Table 2: *Versions of PRESS available for study.*

The other versions cooperate in caching files and differ in terms of their concern with availability, and the performance of their intra-cluster communication protocols.

5 Case Study: Phase 1

We now apply the first phase of our methodology to evaluate the performability of PRESS. We first describe our experimental testbed, then show a sampling of PRESS's behavior under our fault loads. Throughout this section, we do not show results for I-PRESS as they entirely match expectation: the achieved throughput simply depends on how many of the nodes are up and able to serve client requests.

5.1 Experimental Setup

In all experiments, we run a four-node version of PRESS on four 800 MHz PIII PCs, each equipped with 206 MB of memory and 2 10,000 RPM 9 GB SCSI disks. Nodes are interconnected by a 1 Gb/s cLAN network. We can communicate with TCP or VIA over this network. PRESS was allocated 128 MB on each node for its file cache; the remainder of the memory was sufficient for the operating system. In our experiments, PRESS only serves static content and the entire set of documents is replicated at each node on one of the disks. PRESS was loaded at 90% of saturation and set to warm up to this peak throughput over a period of 5 minutes. Note that, because we are running so close to saturation and PRESS already implements sophisticated load balancing, we do not apply a front-end load distributor. Under such high load and little excess capacity, the front-end would not prevent the loss of requests in the event of a fault.

The workload for all experiments is generated by a set of 4 clients running on separate machines connected to PRESS by the same network that connects the nodes of the server. The total network traffic does not saturate any of the cLAN NICs, links, and switch, and so the interference between the two classes of traffic is minimal in our setup. Finally, Mendosus's network emulation system allows us to differentiate between intra-cluster communication and client-server communication when injecting network-related faults. Thus, the clients are never disturbed by faults injected into the intra-cluster communication.

Subsystem	Fault	Characteristics
Network	Link down	Transient - 5, 180 secs
	Switch down	Transient - 5, 180 secs
Disk	SCSI timeout	Transient - 120 secs
	Disk hang	Sticky
	Read faults	Sticky
	Write faults	Sticky
Node	Hard reboot	Transient - 180 secs
	Node freeze	Transient - 180 secs
Application	Process crash	Transient - 180 secs
	Process hang	Transient - 180 secs

Table 3: *Fault loads for PRESS performability study. For transient faults, the given times represent the duration of the faults.*

Each client generates load by following a trace gathered at Rutgers; we chose this trace from several that Carrera and Bianchini previously used to evaluate PRESS's performance because it has the largest working set [7]. Results for other traces are very similar. To achieve a particular load on the server, each client generates a stream of requests according to a Poisson process with a given average arrival rate. Each request is set to time out after 2 seconds if a connection cannot be completed and to time out after 6 seconds if, after successful connection, the request cannot be completed.

Finally, Table 3 lists the set of faults that we inject into a live PRESS system to study its behavior. Faults fall into four categories: network, disk, node, and application. Note that these generic faults can be caused by a wide variety of reasons for a real system; for example, an operator accidentally pulling out the wrong network cable would lead to a link failure. We cannot focus on all potential causes because this set is too large. Rather, we focus on the class of failures as observed by the system, using an MTTF that covers all potential causes of a particular fault. This set is comprehensive with respect to PRESS in that it covers just about all resources that PRESS uses in providing its service.

5.2 Network Faults

In this section, we study PRESS's behavior under network faults. Figure 2 shows the effects of a transient switch fault. We first discuss what happened in each case, then make an interesting general observation.

TCP-PRESS behaved exactly as expected: throughput drops to zero a short time after the occurrence of the

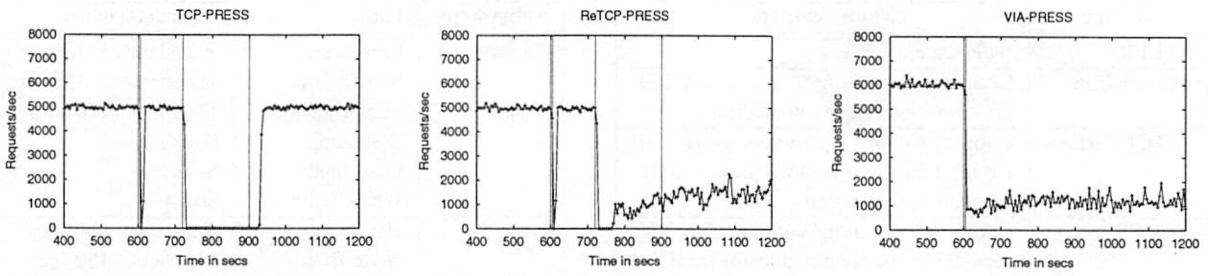


Figure 2: *Effects of transient switch faults. Pairs of vertical lines represent the start and end times of injected faults.*

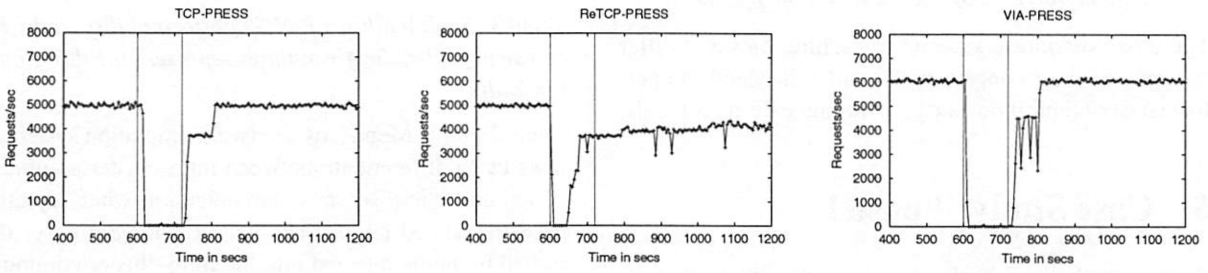


Figure 3: *Effects of transient SCSI time-out faults. Pairs of vertical lines represent the start and end times of injected faults.*

fault because the queues for intra-server communication fill up as the nodes attempt to fetch content across the faulty switch. Throughput stays at zero until the switch comes back up. For ReTCP-PRESS, the first switch fault leads to the same behavior as TCP-PRESS; the reconfiguration code does not activate because the fault is sufficiently short that no heartbeat is lost. The longer switch fault, however, triggers the reconfiguration code, leading ReTCP-PRESS to reconfigure into 4 groups of singleton. The detection time is determined by the heartbeat protocol, which uses a DEADTIME interval of 15 seconds (3 heartbeats). For VIA-PRESS, the switch fault is detected almost immediately by the device driver, which breaks all VIA connections to unreachable nodes. This immediately triggers the reconfiguration of VIA-PRESS into four sub-clusters.

Interestingly, ReTCP-PRESS and VIA-PRESS do not reconfigure back into a single cluster once the switch returns to normal operation. This surprising behavior arises from a mismatch between the fault model assumed by the reconfigurable versions of PRESS and the actual fault. These versions of PRESS assume that nodes fail but links and switches do not. Thus, reconfiguration occurs at startup and on loss of 3 heartbeats, but reintegration into a single group only happens at startup. If a cluster is splintered as above, they never attempt to rejoin. Return to full operation thus would require the intervention of an administrator to restart all but one of the sub-clusters. This, in effect, make these reconfigurable versions *less* robust than the basic TCP-PRESS in the face of relatively short transient faults, and points to the importance of the accuracy of the fault model used in designing a service.

Finally, we do not show results for the link/NIC fault here because they essentially lead to the same behaviors as above.

5.3 Disk Faults

Recall that each server machine contains two SCSI disks, one holding the operating system and the second the file set being served by PRESS. We inject faults into only the second disk to minimize the interference from operating system-related disk accesses (e.g., page swapping) while observing the behavior of PRESS under disk faults.

Figure 3 shows PRESS's behavior under SCSI timeouts. TCP-PRESS and VIA-PRESS behave exactly as one would expect. When the fault lasts long enough, all `disk` helper threads become blocked and the queue between the `disk` threads and the `main` thread fills up. When this happens, the `main` thread itself becomes blocked when it tries to initiate another read. Once one of the nodes grinds to a halt, then the entire server eventually comes to a halt as well. When the faulty disk recovers, the entire system regains its normal operation.

ReTCP-PRESS, on the other hand, interprets the long fault as a node fault and so splinters into sub-clusters, one with 3 nodes and one singleton. This splintering of the

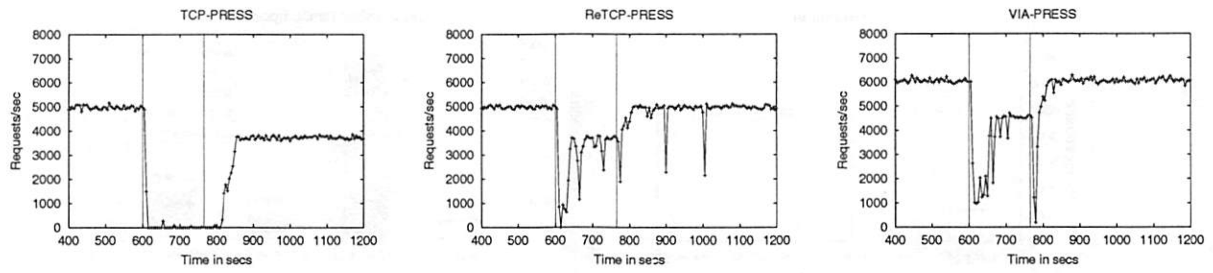


Figure 4: *Effects of a node crash. Pairs of vertical lines represent the start and end times of injected faults.*

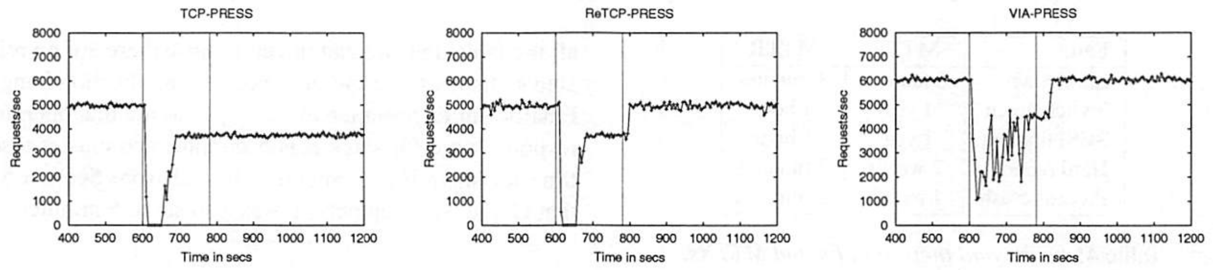


Figure 5: *Effects of one PRESS process crash. Pairs of vertical lines represent the start and end times of injected faults.*

server cluster is caused by missing heartbeats. Similar to the argument for TCP-PRESS and VIA-PRESS above, when all `disk` threads block because of the faulty disk, the `main` thread also eventually blocks when it tries to initiate yet one more read. In this case, however, the main thread is also responsible for sending the heartbeat messages. Thus, when it blocks, its peers do not get any more heartbeats and so assume that that node is down; at this point, the reconfiguration code takes over, leading to the splinter.

We do not show results for disk hang, read and write faults because the behaviors are much as expected.

5.4 Node Faults

Figure 4 shows the effects of a hard reboot fault. Because it is not capable of reconfiguration, TCP-PRESS grinds to a halt while the faulty node is down. When the node successfully reboots, however, the open TCP connections of the three non-faulty nodes with the recovered node break. At this point, the PRESS processes running on these nodes realize that something has happened to the faulty node and stop attempting to coordinate with it. Thus, server operation restarts with a cluster of 3 nodes. When the faulty node successfully completes the reboot sequence, Mendosus starts another PRESS process automatically. However, since TCP-PRESS cannot reconfigure, correct operation with a cluster of 4 nodes cannot take place until the entire server is shutdown and restarted.

ReTCP-PRESS and VIA-PRESS behave exactly as expected. Operation of the server grinds to a halt until the reconfiguration code detects a fault. The three non-faulty nodes recover and operate as a cooperating cluster. When the faulty node recovers and the PRESS process has been restarted, it joins in correctly with the three non-faulty processes and throughput eventually returns to normal.

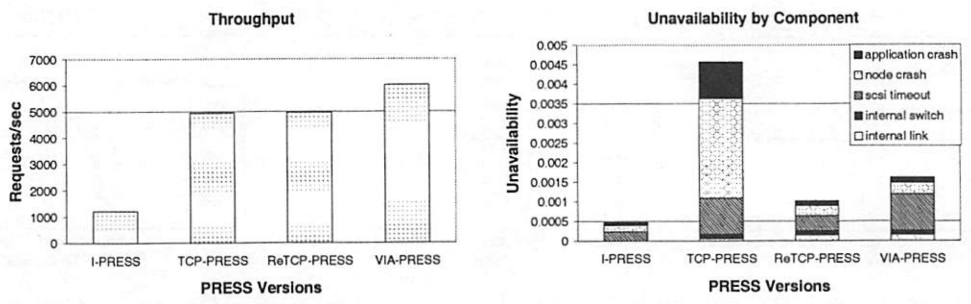
We do not show results for a node freeze because they are similar to those for a SCSI time-out. TCP-PRESS and VIA-PRESS degrades to 0 throughput during the fault but recovers fully. ReTCP-PRESS splinters and cannot recover fully.

5.5 Application Faults

Figure 5 shows the effects of an application crash, which are similar to those of a node crash. The one difference is that TCP-PRESS recovers from 0 throughput more rapidly because it can detect the fault quickly through broken TCP connections. The effects of an application hang are exactly the same as a node freeze.

6 Case Study: Phase 2

We now proceed to the second phase of our methodology: using the analytical model to extrapolate performability from our fault-injection results. We first compare the performance, availability, and performability of the



(a) (b)

Figure 6: *(a) Average modeled throughput and (b) modeled unavailability (1 - availability).*

Fault	MTTF	MTTR
Link down	6 months	3 minutes
Switch down	1 year	1 hour
SCSI timeout	1 year	1 hour
Hard reboot	2 weeks	3 minutes
Process crash	1 month	3 minutes

Table 4: *Faults and their MTTFs and MTTRs.*

Phase	Switch Fault		Application Crash	
	Thruput (reqs/sec)	Duration (secs)	Thruput (reqs/sec)	Duration (secs)
A	892.40	75	1889.10	10
B	–	0	3143.55	145
C	1106.70	3525	4537.60	25
D	–	0	4789.13	45
E	1209.60	300	–	0
F	0.0	300	–	0
G	3017.00	300	–	0

Table 5: *Example throughput and duration of the phases in our model for VIA-PRESS for two different faults. Note that for some types of faults, some phases collapse into a single phase or are not used.*

different versions of PRESS. Then, we show how we can use the model to evaluate design tradeoffs, such as adding a RAID or increasing operator support.

6.1 Parameterizing the Model

We parameterize our model by using the data collected in phase one, the fault load shown in Table 4, and a number of assumptions about the environment. Since we cannot list all data extracted from phase 1 here because of space constraints, we refer the interested reader to http://www.panic-lab.rutgers.edu/Research/mendosus/. Table 5 provides a flavor of this data, listing the throughput and duration of each phase of our 7-stage model for VIA-PRESS for two types of faults. The MTTFs and MTTRs shown in Table 4 were chosen based on previously reported faults and fault rates [13, 16, 32]. Note that we do not model all the faults that we can inject because there are no reliable statistics for some of them, e.g., application hangs. Finally, our environmental assumptions are that operator response time for stage E is 5 minutes and cluster reset time for stage F is 5 minutes. Recall from Section 5.1 that G, the warm up period, was also set to 5 minutes.

6.2 Modeling Results

Figure 6(a) shows the expected average throughput in the face of component faults for the 4 PRESS versions. As has been noted in previous work, the locality-conscious request distribution significantly improves performance. The use of user-level communication improves performance further.

Figure 6(b) shows the average unavailability of the different versions of PRESS. Each bar includes the contributions of the different fault types to unavailability. These results show that availability is somewhat disappointing, on the order of 99.9%, or "three nines". However, recall that the servers were operating near peak; any loss in performance, such as losing a node or splintering, results in an immediate loss in throughput (and in many failed requests). A fielded system would reserve excess capacity for handling faults. Exploring this tradeoff between performability and capacity is a topic for our future research.

Comparing the systems, observe that I-PRESS achieves the best availability because there's no coordination between the nodes. TCP-PRESS is almost an order of magnitude worse than I-PRESS; this is perhaps expected since TCP-PRESS does a very poor job of tolerating and recovery from faults. More interestingly, ReTCP-PRESS gives better availability than VIA-PRESS. Looking at the bars closely, we observe that this is because ReTCP-PRESS is better at tolerating SCSI timeouts. This is fortuitous rather than by design: as previously discussed, when a SCSI timeout occurs, the heartbeats are delayed in ReTCP-PRESS, causing the cluster to reconfigure and proceed without the faulty node. VIA-PRESS does not reconfigure because the

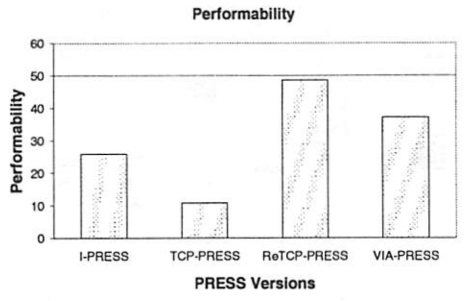


Figure 7: *Performability of each version of PRESS when* $A_I = 0.99999$ *or "five nines availability."*

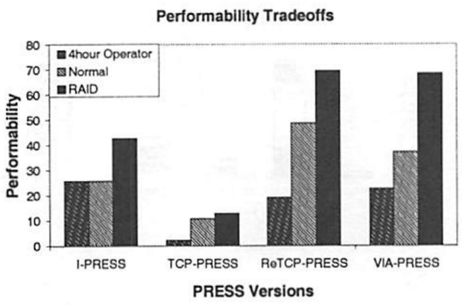


Figure 8: *Impact of reducing the mean operator response from 5 minutes to 4 hours and adding a more reliable disk subsystem.*

communication subsystem does not detect any fault.

Finally, Figure 7 shows the performability of the different PRESS versions. We can see that although the locality-aware, cooperative nature of TCP-PRESS does deliver increased performance, the lack of much, if any, fault-tolerance in its design reduces availability significantly, giving it a lower performability score than I-PRESS. The superior performance of the ReTCP and VIA versions of PRESS make up for their lower availability over I-PRESS. Again, the fact that ReTCP gives a better performability score than VIA-PRESS is the fortuitous loss of heartbeats on SCSI timeouts. Thus, we do not make any conclusion based on this difference (rather, we discuss the nature of heartbeats in Section 7).

Quantifying Design Tradeoffs. Analytical modeling of faults and their phases enables us to explore the impact of our server designs on performability. Thus, we examine two alternative design decisions to the ones we have explored so far.

The first design change is to reduce the operator coverage. In the previous model, the mean time for an operator to respond when the server entered a non-recoverable state was 5 minutes. This represents the PRESS servers running under the watchful eyes of operators 24x7. However, as this is quite an expensive proposition, we reduced the mean response to 4 hours and observed the performability impact.

Figure 6(b) suggests that disks are a major cause of unavailability. In this second design change, we added a much more reliable disk subsystem, e.g., a RAID. We modeled the better disk subsystem by increasing the mean time to failure of the disks by a factor of five, but keeping the MTTR of the disks the same.

Figure 8 shows the performability impact of these two design changes. The center bar represent the same "basic" results as in Figure 7. The left most bar is the basic system with a 4-hour operator response, and the right is the basic system enhanced with a RAID.

Our modeling results show that running the cooperative versions in an environment with quick operator response is critical (unless fault recovery can be improved significantly): the performability of all the cooperative versions become less than that of I-PRESS. On the other hand, our results show that I-PRESS is insensitive to operator response time as expected.

The performability models also demonstrate the utility of a highly-reliable disk subsystem. Figure 8 show that by purchasing increasingly reliable disk subsystems, performability of all versions of PRESS is enhanced, e.g., approximately 84% for VIA-PRESS. In fact, ReTCP-PRESS and VIA-PRESS achieve virtually the same performability in the presence of this more sophisticated disk subsystem. These results suggest that the overall *system impact* of redundant disk organizations, such as RAIDs, is substantial.

Scaling to Larger Cluster Configurations. We now consider how to scale our model to predict the performability of services running on larger clusters. Such scaling may be needed because systems are typically not designed and tested at full deployment scale. We demonstrate the scaling process by scaling our measurements collected on the 4-node cluster to predict PRESS's performability on 8 nodes, which is the largest configuration we can achieve using our current testbed for validation. We then compare these results against what happens if the fault-injections and measurements were performed directly on an 8-node system.

Essentially, our model depends on three types of parameters: the mean time to failure of each component ($MTTF_c$), the duration of each phase during the fault (D_c^p), and the (average) throughput under normal operation (T_n) and during each fault phase (T_c^p). These parameters are affected by scaling in different ways.

Let us refer to the $MTTF_c$ in a configuration with N nodes as $MTTF_c^N$. $MTTF_c$ in a configuration with S times more nodes, $MTTF_c^{SN}$, is $MTTF_c^N/S$. Assuming that the bottleneck resource is the same for the N-node and the SN-node configurations, the durations D_c^p should be the same under both configurations. Further

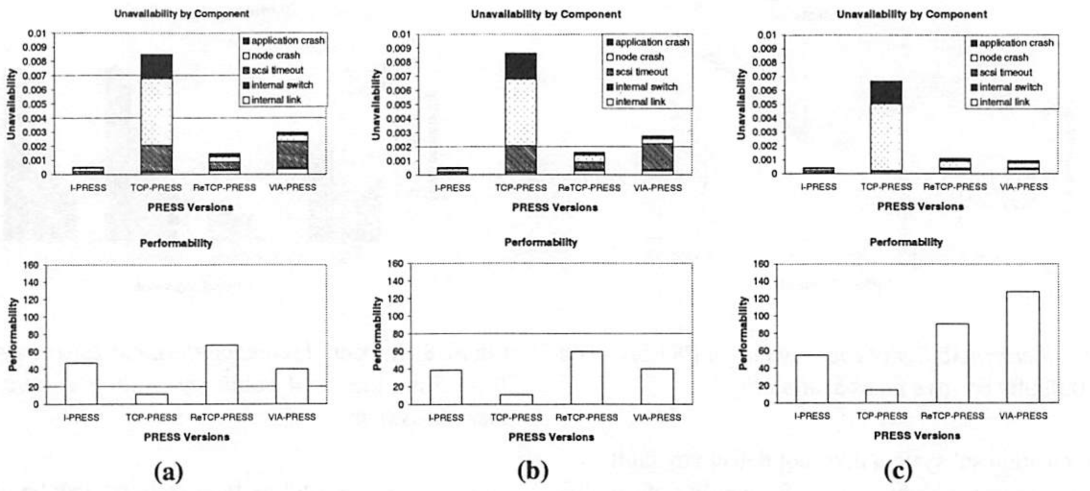


Figure 9: *Unavailability and performability of PRESS on 8 nodes, when (a) scaling analytically from a 4-node configuration in which PRESS has 128 MB of memory on each node; (b) using measurements on 8 nodes, each of which has 64 MB of memory; and (c) using measurements on 8 nodes, each of which has 128 MB of memory.*

assuming a linear increase in throughput for PRESS (in general, designers using our methodology will need to understand/measure throughput vs. number of nodes for the service under study for scaling), we scale throughput as $T_n^{SN} = S \times T_n^N$. Unfortunately, the effect of scaling on the throughput of each fault phase is not as straightforward. When the effect of the fault is to bring down a node or make it unaccessible, for instance, the throughput of phase C should approach $T_n^{SN} - T_n^{SN}/SN$ under SN nodes, whereas the average throughput of phases B and G should approach $(T_n^{SN} - T_n^{SN}/SN)/2$ and $T_n^{SN}/2$ respectively. Just as under N nodes, the throughput of phases A and F should approach 0 under SN nodes.

Finally, an important effect occurs as we scale up, when the cluster-wide cache space almost eliminates accesses to disks. This reduces the impact of disk faults on availability, because the duration of a fault may not overlap with any accesses to cold files.

Figure 9 shows the scaled modeling results using the above rules, as well as results using measurements taken directly on PRESS running on 8-nodes. Observe that our scaling results are very accurate compared to the measured results for 8 nodes, each with 64 MB of memory. This is because the cluster-wide amount of cache memory was kept constant, thus, PRESS still depended on the disk for fetching cold files. However, if each node in the 8-node cluster has 128 MB, then the results are significantly different. Because disk faults no longer impact availability—the working set fits in memory—VIA-PRESS now achieves the best performability. When we eliminate the contribution of disk faults to unavailability in our scaled modeling, we again achieve a close match between modeling and measurements.

In summary, we observe that our methodology produces accurate results when scaling to larger configurations than that available at design/testing time. However, the service designer must understand the system well enough to account for effects of crossing boundaries, where some resource may become more or less critical to system behavior.

7 Lessons Learned

Applying our 2-phased methodology to PRESS we learned a few lessons. First, we found that the fault-injection phase of our methodology exposed not only implementation bugs, but more importantly conceptual gaps in our reasoning about how the system reacts to faults. For example, an intermittent VIA switch fault created a network of singletons that was incapable of rejoining, even after the switch came back on-line. These experiments demonstrated that some of the faults not originally considered in the PRESS design had a significant impact on the behavior of the server. We also found the second phase of the methodology to be extremely useful for quantitatively reasoning about the impact of design tradeoffs. For example, for all versions except I-PRESS, operator response time is critical to overall performability. While a designer may intuitively understand this, our methodology allows us to quantify the impact of decreasing or increasing operator support.

The second lesson is that runtime fault detection and diagnosis is a difficult issue to address. Consider the heartbeat system implemented in ReTCP-PRESS. What should a loss of heartbeat indicate? Should it indicate a node fault? Does it indicate *some* failure on the node? How can we differentiate between a node and a communication fault? Should we differentiate between node and application faults? Again, this implies that systems must carefully monitor the status of all its components, as well as have a well-defined reporting system, in which each status indicator has a clearly defined semantic.

Finally, efforts to achieve high availability will only pay off if the assumed fault model matches actual fault scenarios well. Mismatches between PRESS's fault model and actual faults led to some surprising results. A prime example of this is PRESS's assumption that the only possible faults are node or application crashes. This significantly degrades the performability of ReTCP-PRESS and VIA-PRESS because other faults that also led to splintering of the cluster (e.g., link fault) eventually required the intervention of a human operator before full recovery could occur.

One obvious answer to this last problem is to improve PRESS's fault model, which is currently very limited. However, the more complex the fault model, the more complex the detection and recovery code, leading to higher chances for bugs. Further, detection would likely require additional monitoring hardware, leading to higher cost as well. One idea that we have recently explored in [24, 26] is to define a limited fault model and then to enforce that fault model during operation of the server. We refer to this approach as *Fault Model Enforcement* (FME). As an example FME policy, in [24] we enforced the node crash model in PRESS by forcing any fault that leads to the separation of a process/node from the main group to cause the automatic reboot of that node. While this is an extreme example of FME, it does improve the availability of PRESS substantially, as well as reduces the need for operator coverage.

8 Related Work

There has been extensive work in analyzing faults and how they impact systems [11, 31, 17]. Studies benchmarking system behavior under fault loads include [15, 19]. Unfortunately, these works do not provide a good understanding of how one would estimate overall system availability under a given fault load.

There has also been a large number of system availability studies. Two approaches that are used most often include empirical measurements of actual fault rates [3, 13, 20, 16, 23] and a rich set of stochastic process models that describe system dependencies, fault likelihoods over time, and performance [10, 21, 30]. Compared to these complex stochastic models, our models are much simpler, and thus more accessible to practitioners. This stems from our more limited goal of quantifying performability to compare systems, as opposed to reasoning about system evolution as a function of time.

A recent work [1] proposed that faults are unavoidable and so systems should be built to recover rapidly, in addition to being fault-tolerant. While similar in viewpoint, our proposed methodology concentrates more on evaluating performability independently of the approach taken to improve performance or availability.

Perhaps more similar to our work is that of [6], which outlines a methodology for benchmarking systems' availability. Other works have proposed robustness [29] and reliability benchmarks [34] that quantify the degradation of system performance under faults. Our work here differs from these previous studies in that we focus on cluster-based servers. Our methodology and infrastructure seem to be the first directed to studying these servers, although recent studies have looked at response time and availability of a single-node Apache Web server [18]. Though other previous work proposes availability-improving strategies for applications spanning large configurations [9], we seem to be the first group to quantify the performability and the design and environmental tradeoffs of cluster-based servers.

Finally, we recently used the methodology introduced here to quantify the effects of two different communication architectures on the performability of PRESS [25].

9 Conclusions

The need for appropriate methodologies and infrastructures for the design and evaluation of highly available servers is rapidly emerging, as availability becomes an increasingly important metric for network services. In this paper, we have introduced a methodology that uses fault-injection and analytical modeling to quantitatively evaluate the performance *and* availability (performability) of cluster-based services. Designers can use our methodology to study *what if* scenarios, predicting the performability impact of design changes. We have also introduced Mendosus, a fault-injection and network emulation infrastructure designed to support our methodology.

We evaluated the performability of four different versions of PRESS, a sophisticated cluster-based server, to show how our methodology can be applied. In addition, we also showed how our methodology can be used to assess the potential impact of different design decisions and environmental parameters. An additional benefit of studying the various versions of PRESS is that our results provided insights into server design, particularly concerning runtime fault detection and diagnosis.

References

[1] P. D. A., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Op-

penheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, March 2002.

[2] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-Based Network Servers. In *Proceedings of USENIX'2000 Technical Conference*, San Diego, CA, June 2000.

[3] S. Asami. Reducing the cost of system administration of a disk storage system built from commodity components. Technical Report CSD-00-1100, University of California, Berkeley, June 2000.

[4] R. Bianchini and E. V. Carrera. Analytical and Experimental Evaluation of Cluster-Based WWW Servers. *World Wide Web Journal*, 3(4):215–229, December 2000.

[5] E. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, July/August 2001.

[6] A. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA*, June 2000.

[7] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[8] E. V. Carrera, S.Rao, L.Iftode, and R. Bianchini. User-Level Communication in Cluster-Based Servers. In *Proceedings of the Proceedings of the 8th IEEE International Symposium on High-Performance Computer Architecture (HPCA 8)*, February 2002.

[9] A. Fox and E. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of Hot Topics in Operating Systems (HotOS VII)*, Mar. 1999.

[10] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. Analysis of preventive maintenance in transactions based software systems. *IEEE Transactions on Computers*, 47(1):96–107, Jan. 1998.

[11] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, Oct. 1990.

[12] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 319–332, Oct. 2000.

[13] T. Heath, R. Martin, and T. D. Nguyen. Improving Cluster Availability Using Workstation Validation. In *Proceedings of the ACM SIGMETRICS 2002*, Marina Del Rey, CA, June 2002.

[14] G. W. Herbert. Failure from the Field: Complexity Kills. In *Proceedings of the Second Workshop on Evaluating and Architecting System dependabilitY (EASY)*, Oct. 2002.

[15] P. J. K. Jr., J. Sung, C. P. Dingman, D. P. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Symposium on Reliable Distributed Systems*, pages 72–79, 1997.

[16] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proceedings of the 18th Symposium on Reliable and Distributed Systems (SRDS '99)*, 1999.

[17] I. Lee and R. Iyer. Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system. In *Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, pages 20–29, 1993.

[18] L. Li, K. Vaidyanathan, and K. S. Trivedi. An Approach for Estimation of Software Aging in a Web Server. In *Proceedings of the International Symposium on Empirical Software Engineering, ISESE 2002*, Nara, Japan, Oct. 2002.

[19] T. Liu, Z. Kalbarczyk, and R. Iyer. A software multilevel fault injection mechanism: Case study evaluating the virtual interface architecture. In *Symposium on Reliable Distributed Systems*, 1999.

[20] D. D. E. Long, J. L. Carroll, and C. J. Park. A Study of the Reliability of Internet Sites. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 177–186, Pisa, Italy, Sept. 1991.

[21] J. F. Meyer. Performability evaluation: Where it is and what lies ahead. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 334–343, Erlangen, Germany, Apr. 1995.

[22] B. Murphy and T. Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, pages 341–353, 1995.

[23] B. Murphy and B. Levidow. Windows 2000 Dependability. Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.

[24] K. Nagaraja, R. Bianchini, R. Martin, and T. D. Nguyen. Using Fault Model Enforcement to Improve Availability. In *Proceedings of the Second Workshop on Evaluating and Architecting System dependabilitY (EASY)*, Oct. 2002.

[25] K. Nagaraja, N. Krishnan, R. Bianchini, R. P. Martin, and T. D. Nguyen. Evaluating the Impact of Communication Architecture on the Performability of Cluster-Based Services. In *Proceedings of the 9th Symposium on High Performance Computer Architecture (HPCA-9)*, Annaheim, CA, Feb. 2003.

[26] K. Nagaraja, N. Krishnan, R. Bianchini, R. P. Martin, and T. D. Nguyen. Quantifying and Improving the Availability of Cooperative Cluster-Based Services. Technical Report DCS-TR-517, Department of Computer Science, Rutgers University, Jan 2003.

[27] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, October 1998.

[28] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, Charlston, SC, Dec. 1999.

[29] D. Siewiorek, J. Hudakund, B. Suh, and Z. Segall. Development of a benchmark to measure system robustness. In *In Proceedings 23rd International Symposium Fault-Tolerant Computing*, pages 88–97, 1993.

[30] R. M. Smith, K. S. Trivedi, and A. V. Ramesh. Performability Analysis: Measures, an Algorithm, and a Case Study. *IEEE Transactions on Computers*, 37(4), April 1998.

[31] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[32] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *The 1999 Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[33] D. Tang and R. K. Iyer. Analysis and Modeling of Correlated Failures in Multicomputer Systems. *IEEE Transactions on Computers*, 41(5):567–577, May 1992.

[34] T. K. Tsai, R. K. Iyer, and D. Jewitt. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.

[35] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *1999 Pacific Rim International Symposium on Dependable Computing*, Dec. 1999.

Mayday: Distributed Filtering for Internet Services

David G. Andersen
MIT Laboratory for Computer Science
dga@nms.lcs.mit.edu

Abstract

Mayday is an architecture that combines overlay networks with lightweight packet filtering to defend against denial of service attacks. The overlay nodes perform client authentication and protocol verification, and then relay the requests to a protected server. The server is protected from outside attack by simple packet filtering rules that can be efficiently deployed even in backbone routers.

Mayday generalizes earlier work on Secure Overlay Services. Mayday improves upon this prior work by separating the overlay routing and the filtering, and providing a more powerful set of choices for each. Through this generalization, Mayday supports several different schemes that provide different balances of security and performance, continuum, and supports mechanisms that achieve better security or better performance than earlier systems. To evaluate both Mayday and previous work, we present several practical attacks, two of them novel, that are effective against filtering-based systems.

1 Introduction

Denial of service (DoS) attacks are potentially devastating to the victim and require little technical sophistication or risk exposure on the part of the attacker. These attacks typically attempt to flood a target with traffic to waste network bandwidth or server resources. To obtain the network bandwidth necessary to attack well-connected Internet services, attackers often launch Distributed DoS (DDoS) attacks, where tens to thousands of hosts concurrently direct traffic at a target. The frequency of these attacks is startling—one analysis of attack "backscatter" suggests that hundreds of these attacks take place each day [19]. DDoS attacks no longer require a high degree of sophistication. So-called "rootkits" are available in binary form for a variety of platforms, and can be deployed using the latest off-the-shelf exploits. Even worm programs have been used to launch DDoS attacks [7].

While technical measures have been developed to prevent [12, 20, 15] and trace [27, 10, 24] DDoS attacks, most of these measures require wide-spread adoption to be successful. Unfortunately, even the simplest of these measures, filtering to prevent IP address spoofing, is not globally deployed despite years of advocacy. While there is some interim benefit from the incremental deployment of earlier measures, they lack the deployment incentive of a solution that provides immediate relief to the deployer.

An ideal DDoS prevention system stops attacks as close to their source as possible. Unfortunately, the targets of attacks have the most incentive to deploy solutions, and deployment is easiest inside one's own network. Intrusive systems that perform rate-limiting or that require router modifications hold promise, but most Internet Service Providers (ISPs) are unwilling or unable to deploy these solutions in the places where they would be most effective—in their core or at their borders with other ISPs.

We study a set of solutions that are more resource-intensive to deploy, because they require overlay nodes, but that are easily understood and implemented by ISPs using conventional routers. Trace-based *reactive* solutions impose no overhead during normal operation, but suffer from a time lag before recovering from an attack. Our solution, an architecture called Mayday, provides *pro-active* protection against DDoS attacks, imposing overhead on all transactions to actively prevent attacks from reaching the server. Mayday generalizes the Secure Overlay Services (SOS) approach [18]. Mayday

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933. David G. Andersen is supported by a Microsoft Graduate Fellowship.

uses a distributed set of **overlay nodes** that are trusted (or semi-trusted) to distinguish legitimate traffic from attack traffic. To protect a server from DDoS traffic, Mayday prevents general Internet hosts from communicating directly with the server by imposing a router-based, network-layer **filter ring** around the server. Instead, clients communicate with the overlay nodes, who verify that the client is permitted to use the service. These overlay nodes then use an easily implemented **lightweight authenticator**, such as sending the traffic to the correct TCP port on the server, to get through the filter ring. Within this framework, SOS represents a particular choice of authenticator and overlay routing, using distributed hash table lookups to route between overlay nodes, and using the source address of the overlay node as the authenticator. We explore how different organizations of the authentication agents operate under various threat models, and present several lightweight authenticators that provide improved levels of defense over source address authentication.

Finally, we define several threat models with which we evaluate pro-active DDoS protection. Within these threat models, we present several attacks, including a novel scanning method we call next-hop scanning, that are effective against SOS, certain variants of Mayday, and against conventional router-based filtering of DDoS attacks.

2 Related Work

DoS flooding attacks have been well studied in the recent literature. Most work in this area has been aimed at either preventing attacks by filtering, or at detecting attacks and tracing them back to their origin. Overlay networks have been used in many contexts to speed deployment of new protocols and new functionality.

2.1 Attack Prevention

The most basic defense against anonymous DoS attacks is *ingress filtering* [11]. Ingress filtering is increasingly deployed at the edge of the network, but its deployment is limited by router resources and operator resources. Ingress filtering also interferes with Mobile IP techniques and split communication systems such as unidirectional satellite systems. Despite these limitations, in time, address filtering should become widespread, enhanced by mechanisms such as Cisco's Reverse Path Filtering. However, ingress filtering is most effective at the edge; deployment in the core, even if it becomes technically feasible, is not completely effective [20].

Mazu Networks [1] and Arbor Networks [4] provide DoS detection and prevention by creating models of "normal" traffic and detecting traffic that violates the model. If an attack is detected, Mazu's tools suggest access lists for routers. If the Mazu box is installed in-line with the network, it can shape traffic to enforce a previously good model. Asta Networks' Vantage analyzes NetFlow data to detect DoS attacks on high-speed links and suggest access lists to staunch the flood [5]. These access lists must be deployed manually, and provide reactive, not proactive, assistance to the victim of a DoS attack. Because these schemes result in access lists being applied at routers, many of the probing attacks we discuss in Section 4 can be used against these solutions as well.

Pushback provides a mechanism for pushing rate-limiting filters to the edges of an ISP's network [15]. If attack packets can be distinguished from legitimate traffic (as in the case of a SYN flood), Pushback's mechanisms can effectively control a DoS attack. In the general case, Pushback will also rate-limit valid traffic. If the source of the traffic is widely distributed over the network, Pushback is less effective. In any event, Pushback is effective at reducing collateral damage to other clients and servers that share links with the DoS target, but this scheme requires new capabilities of routers, slowing deployment.

2.2 Attack Detection

ICMP traceback messages were proposed as a first way of tracing the origins of packets [6]. Under this scheme, routers would periodically send an ICMP message to the destination of a packet. This message would tell the recipient the link on which the packet arrived and left, allowing the recipient of a sufficient quantity of ICMP traceback messages to determine the path taken by the packets.

To avoid out-of-band notifications, Savage et al. use probabilistic inline packet marking to allow victims to trace attack packets back to their source [24]. In this scheme, routers occasionally note in the packet the link the packet has traversed; after sufficient packets have been received by the victim host, it can reconstruct the full path taken by the packets. Dean et al., treat the path reconstruction problem as an algebraic coding problem [10]. These refinements improve the performance and robustness of the packet marking, but the underlying technique is similar to the original.

The probabilistic traceback schemes require that a large amount of data be received by a victim before path reconstruction can be performed. To allow traceback of even a single packet, the Source Path Isolation Engine

(SPIE) system records the path taken by *every* packet that flows through a router [27]. SPIE uses a dense bloom-filter encoding to store this data efficiently and at high speeds. While it provides exceptional flexibility, SPIE requires extensive hardware support.

2.3 Overlay Networks

Overlay networks have long been used to deploy new features. Most relevant to this work are those projects that used overlays to provide improved performance or reliability. The Detour [23] study noted that re-routing packets between hosts could often provide better loss, latency, and throughput than the direct Internet path. The RON project experimentally confirmed the Detour observations, and showed that an overlay network that performs its own network measurements can provide improved reliability [2].

Content Delivery Networks such as Akamai [31], and Cisco's Overcast [16] use overlay networks to provide faster service to clients by caching or eliminating redundant data transmission. The ideas behind these networks would integrate well with Mayday; in fact, the Akamai network of a few thousand distributed nodes seems like an ideal environment in which to deploy a Mayday-like system.

Mixnet-based anonymizing overlays like Tarzan [13] are designed to prevent observers from determining the identity of communicating hosts. The principles used in these overlays, primarily Chaumian Mixnets [8], can be directly used in a system such as Mayday to provide greater protection against certain adversaries. We discuss this further in Section 3.5.

3 Design

The design of Mayday evolved from one question: Using existing network capabilities, how do we protect a server from DDoS attacks while ensuring that legitimate clients can still use the services it provides? To answer this question, we restricted ourselves to using only routers with limited packet filtering abilities, or more powerful hosts that aren't on the forwarding path. We wish to provide protection against realistic attackers who control tens or thousands of machines, not malicious network operators or governments. Before exploring the design of our system, we define these attackers and the capabilities they possess. For this discussion, the *server* is a centralized resource that is required in order to provide some service. *Clients* are authorized to use the service, but are not trusted to communicate directly with the server because clients are more numerous and more prone to compromise. *Overlay nodes* are hosts scattered around the Internet that act as intermediaries between the clients and the server.

3.1 Attacker Capabilities

DDoS attacks can be mounted with a relatively low degree of technical sophistication. We focus exclusively on *flooding* attacks, and not on attacks that could crash services with incorrect data. (Using the overlay nodes as protocol verifying agents could prevent some data-based attacks as well.) The simplest flooding attacks (which may be effective if launched from a well-connected site) require only a single command such as `ping`. Many sophisticated attacks come pre-packaged with installation scripts and detailed instructions, and can often be used by people who may not even know how to program. The greatest threat to many Internet services comes from relatively simple attacks because of their ease of use and ready accessibility. We therefore concentrate on simpler attacks.

We assume that all attackers can send a large amount of data in arbitrary formats from forged source addresses of their choice. Ingress filtering may reduce the number of hosts with this capability, but is unlikely to eliminate all of them. Certain attackers may have more resources available, may be able to sniff traffic at points in the network, and may even be able to compromise overlay nodes. We consider the following classes of attackers:

The **Client Eavesdropper** can view the traffic going to and from one or more clients, but cannot see traffic that has reached an overlay node or the target.

The **Legitimate Client Attacker** is authorized to use the service, or is in control of an authorized client.

The **Random Eavesdropper** can monitor the traffic going to one or more overlay nodes, but cannot choose which overlay nodes are watched.

The **Targeted Eavesdropper** can view the traffic going to and from any particular overlay node, but not all overlay nodes at once (i.e., changing monitored nodes requires non-negligible time).

The **Random Compromise Attacker** can compromise one or more randomly chosen overlay nodes.

The **Targeted Compromise Attacker** can select a particular overlay node, or a series of them, and obtain full control of the node.

We ignore certain attackers. For instance, an attacker capable of watching all traffic in the network, or compromising all nodes concurrently, is too powerful for our model to resist. The difference between these global attackers and the targeted eavesdropper or compromiser is one of time and effort. Given sufficient time, the targeted compromise attacker may be able to control all nodes, but during this time the service provider may counteract the offense.

3.2 Mayday Architecture

The Mayday architecture assumes that some entity—perhaps the server's ISP—has routers around the server that provide its Internet connectivity, and is willing to perform some filtering at those routers on behalf of its client. We term this set of routers the *filter ring*. While the ring could be implemented by filtering at the router closest to the server, this would provide little attack protection, because the traffic would consume the limited bandwidth close to the server. Pushing the filter ring too far from the server increases the chance that nodes inside the filter can be compromised. Instead, the ring is best implemented near the core-edge boundary, where all traffic to the server passes through at least one filtering router, but before the network bottlenecks become vulnerable to attack. To provide effective protection against large attacks, this filtering must be lightweight enough to be implemented in high-speed core routers.

The requirement for a fast router implementation rules out certain design choices. One obvious mechanism would be for clients to use IPSec to authenticate themselves to a router in the filter ring, at which point the router would pass the client's traffic through. If a service provider is capable of providing this service, along with rate limiting, a server should be well-protected from DoS attacks.

The Mayday architecture is designed to work with more limited routers. Modern routers can perform routing lookups very quickly, and many (but not all) can perform a *few* packet filtering operations at line speed. Clients, however, may be many in number, or the set of clients may change dynamically. Client verification may involve database lookups or other heavyweight mechanisms. Access lists in core routers are updated via router configuration changes, so network operators are not likely to favor a solution that requires frequent updates. Creating an access list of authorized clients is probably not practical due to client mobility and the sheer size of such a list; we need filter keys that change less often. We term these filter keys the *lightweight authenticators*.

To handle the joint requirements of client authentication and feasible implementation, we add a fourth type of party, the *overlay nodes*. Clients talk directly to an overlay node, the *ingress node*, not to the server or filter ring. Some of the overlay nodes, the *egress nodes*, can talk to the server through the filter ring. If the ingress node is not also an egress node, the request must be routed through the overlay to an egress node. Figure 1 shows the general Mayday architecture.

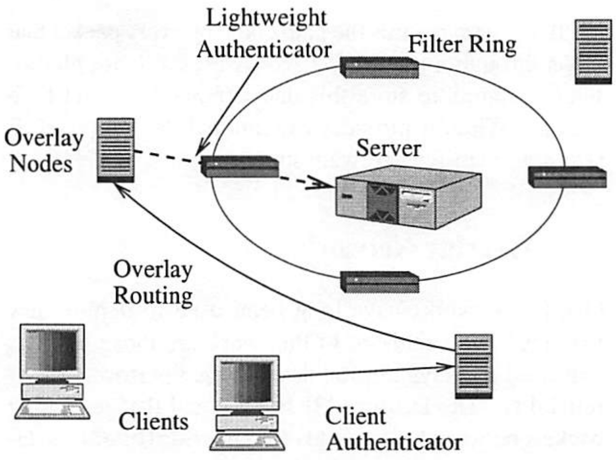


Figure 1: The Mayday architecture. Clients communicate with overlay nodes using an application-defined client authenticator. Overlay nodes authenticate the clients and perform protocol verification, and then relay requests through the filter ring using a lightweight authenticator. The server handles requests once they pass through the network-layer filter ring.

Using this architecture, a designer can make several choices to trade off security, performance, and ease of deployment. First, the designer can first pick one of several overlay routing methods: more secure overlay routing techniques reduce the impact of compromised overlay nodes, but increase request latency. Second, the designer can pick one of several lightweight authenticators, such as source address or UDP/TCP port number. The choice of authenticator affects both security and the overlay routing techniques that can be used. The security and performance of the resulting system depend on the *combination* of authenticator and overlay routing. We discuss the properties of these combinations in Section 3.6 after describing the individual mechanisms.

3.3 Client Authentication

Clients must authenticate themselves to the overlay before they are allowed to access the server. The nature of the client authentication depends on the service being protected. If Mayday is used to protect a small, private service, clients could be authenticated using strong cryptographic verification. In contrast, if Mayday is protecting a large, public service such as Yahoo!, client authentication may be only a database verification of the user's password. Mayday leaves client authentication up to the system designer, since it is inextricably linked to the specific application being protected.

3.4 Lightweight Authenticators

Mayday uses lightweight authentication tokens to validate communication between the overlay node(s) and the server. Mayday requires its tokens be supported with low overhead by commodity routers. Modern routers can filter on a variety of elements in the packet header, such as source and destination address, UDP or TCP port number, and so on. Several of these fields can be used as authenticators. All "source" addresses and ports refer to the egress node; "destination" addresses and ports refer to the server. Each of these fields has its own strengths and weaknesses as a lightweight authenticator:

- **Egress Source Address**: Source filtering is well understood by network operators, and gains effectiveness when other providers deploy IP spoofing prevention. It limits the number of overlay nodes that can communicate with the server. SOS uses this authenticator.
- **Server Destination Port**: The UDP or TCP destination port is an obvious key to use. If the overlay network has fewer than 65,000 nodes, this key provides a larger space in which an attacker must search to get through the firewall. It allows multiple authorized sources to communicate with the server. In other respects, it is similar to source address authentication. The source port can also be used, but this limits the total number of concurrent connections to the server.
- **Server Destination Address**: If the server has a variety of addresses that it can use, the destination address can be used as an authentication token. For example, if a server is allocated the netblock 192.168.0.0/24, its ISP would announce this entire block to the world. Internally, the server would only announce a single IP address to its ISP, and send a null route for the remaining addresses. Thus, a packet to the correct IP would go to the server, but packets to the other IP addresses would be dropped at the border routers. The advantage of this mechanism is that it requires no active support from the ISP to change filters, and uses the fast routing mechanisms in routers, instead of possibly slower filtering mechanisms[1]. Because it uses standard routing mechanisms, updates could be pushed out much more rapidly than router reconfigurations for filter changes. We term this effect *agility*, and discuss later how it can provide freshness to authenticators. The disadvantage is that it wastes address space (a problem solved by IPv6, though IPv6 has its own deployment delays). Destination address filtering is unique in that it can be changed very dynamically by routing updates, even in a large network of routers.
- **Other header fields**: Some routers can filter on attributes like protocol, packet size, and fragment offset. Each of these fields can be manipulated by the egress node to act as a lightweight authenticator, but they require lower-level hacks to set. While they could be useful for providing additional bits of key space, they are less usable than port or address filtering, except to provide some security through obscurity.
- **User-defined fields**: Firewalls can filter on user-defined fields inside packets. This approach provides a huge keyspace and source address flexibility, but few core routers support this feature.

Authentication tokens can be combined. Using both source address and port verification provides a stronger authenticator than source address alone, making some of the attacks we discuss in section 4 difficult to pull off.

3.5 Overlay Routing

The choice of overlay routing can reduce the number of overlay nodes that have direct access to the server, thus providing increased security. These choices can range from direct routing, in which every overlay node can directly access the server (i.e. be an egress node), to Mixnet-style routing ("onion routing"), in which the other overlay nodes do not know which is the egress node [13].

The choice of lightweight authenticator affects which overlay routing techniques can be used. For instance, using source address authentication with proximity routing is extremely weak, because an attacker already knows the IP addresses of the overlay nodes, and any of those addresses can pass the filter.

- **Proximity Routing**: By picking the overlay node nearest the client (similar to Akamai and other CDNs [31]) or the node that provides the best performance between client and server [2, 16], the system can provide high performance with low

[1]An interesting manual use of destination filtering occurred during the Code Red worm in 2001. The worm was designed to flood the IP address of `www.whitehouse.gov`, but had a hardcoded address, not a DNS lookup. The site administrators changed the service address and filtered the old address to successfully protect themselves from the attack.

overhead. In fact, when combined with overlay-level caching, this design could provide better performance than direct client-server communication. Proximity routing requires that all overlay nodes possess the lightweight authenticator.

- **Singly-Indirect Routing**: The ingress node passes the message directly to the egress node, which sends the message to the server. All overlay nodes know the identity of the egress node.
- **Doubly-Indirect Routing**: Ingress nodes send all requests to one or more overlay nodes, who then pass the traffic to the egress node. Only a subset of overlay nodes know the identity of the egress node. SOS uses this scheme.
- **Random Routing**: The message is propagated randomly through the overlay until it reaches a node that knows the lightweight authenticator. Adds $O(N)$ additional overlay hops, but provides better compromise containment. In most ways, this routing is inferior to mix routing.
- **Mix Routing**: Based on Mixnets [8] and the Tarzan [13] anonymous overlay system. A small set of egress nodes configure encrypted forwarding tunnels through the other overlay nodes in a manner such that each node knows only the next hop to which it should forward packets, not the ultimate destination of the traffic. At the extreme end of this style, *cover traffic*—additional, fake traffic between overlay nodes—can be added to make it difficult to determine where traffic is actually originating and going. This difficulty provides protection against even the targeted eavesdropper and compromise attacker, but it requires many overlay hops and potentially expensive cover traffic.

3.6 Choice of Authenticator and Routing

The major question for implementation is which combination of overlay routing and authenticator to use. An obvious first concern is practicality: If a service provider is only able to provide a certain type of filtering, the designer's choices are limited. There are three axes on which to evaluate the remaining choices: performance, security, and agility. Many combinations of authenticator and routing fall into a few "best" equivalence classes that trade off security or performance; the remaining choices provide less security with the same performance, or vice versa.

High performance: Proximity routing provides the best performance, but is vulnerable to the random eavesdropper. Works with any authenticator *except* source address, since the address of the overlay nodes is known. Blind DoS attacks against the system are difficult, since all nodes can act as ingress and egress nodes. Singly-indirect routing with source address provides equivalent protection with inferior performance.

Eavesdropping Resistance, Moderate Performance: Singly-indirect routing, when used with any authenticator other than source address, provides resistance to the random eavesdropper and random compromise attack, because only a small number of nodes possess the authentication key.

SOS: The SOS method uses doubly-indirect routing with source address authentication. In the SOS framework, packets enter via an "access node," are routed via a Chord overlay [29] to a "beacon" node, and are sent from the "beacon" node to the "servlet" node. The servlet passes packets to the server. This method provides equivalent security to the singly-indirect scheme above, but imposes at least one additional overlay hop.

Agility: singly-indirect routing with destination address authentication provides an agile (and deployable) system. Because routing updates, not manual configuration changes, are used to change the lightweight authenticator, it is feasible to update the authentication token often. This agility can be used to resist adaptive attacks by changing the authentication token before the attack has sufficiently narrowed in on the token. Destination address authentication can provide this benefit in concert with other authenticators (such as port number) to provide an agile scheme with a large number of authenticators.

Maximum Security: By using Mix-style routing with cover traffic, a service provider can provide some resistance against the targeted compromise attacker (With 3-hop Tarzan routing, an attacker must compromise 24 nodes to reach the egress node). By using agile destination-address based authentication, the service provider gains resistance to adaptive attacks. By combining the agile authenticator with port number authentication, the system increases its key space, while retaining the ability to recover from egress node failures. Alternately, source address authentication would slow this recovery, but it practically reduces the number of attack nodes that can successfully be used since many Internet hosts are filtered.

3.7 Switchable Protection

By using destination address-based filtering, we can provide *switchable* DoS protection: When no attack is present, clients may directly access the service. When an attack commences, the system can quickly and automatically switch to a more secure mode, assuming that some channel exists to notify the nodes and routers of the change. This allows us to use Mayday as both a reactive and a proactive solution.

The service provider is given two or more IP addresses. IP address A is the "normal" mode address, and the other addresses are the "secure" mode addresses. When an attack commences, the service sends a routing update (in the same manner as destination-address based authentication) to change to one of the secure addresses.

The limiting step in reactive DoS protection is configuring access lists at many routers concurrently. To speed this step, the ISP configures two sets of access lists *in advance*. The first list permits access from all clients (or all overlay nodes, for proximity routing) to the normal mode address A. The second list restricts access with a lightweight authenticator, and applies to the secure mode addresses. The server can quickly switch modes by sending a routing update.

This scheme works best when normal mode is proximity routing through all overlay nodes, and secure mode involves more stringent routing and filtering. In this case, the addresses to which clients connect do not change, and client connections need not be interrupted at the commencement of a DDoS attack. If a brief interruption is tolerable, a DNS update can be pushed out to point new connections to the overlay nodes.

3.8 Changing Authenticators or Overlay Nodes

Changing the authentication key or the overlay nodes through which traffic passes could break currently open connections. Fortunately, the communication between the ingress node and the server is completely under the control of the system designer. When a connection between the ingress node and server is interrupted by address or port changes, the designer can use mechanisms such as TCP Migrate [26] or other end-to-end mobility solutions to transparently reconnect the session. Using these mechanisms between ingress node and server would not require client changes.

4 Attacks and Defenses

The ability of an overlay-based architecture to resist simple flooding attacks was explored in the SOS study. For various simple attack models, a sufficiently large number of overlay nodes can resist surprisingly strong attacks targeted against the server or against individual overlay nodes. In this section, we examine a more sophisticated

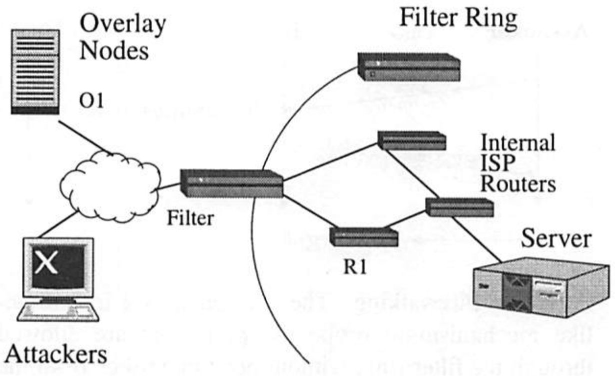


Figure 2: The framework for considering attacks. Overlay nodes and clients are both outside the filter ring. Inside the filter ring may be more ISP routers, which eventually connect to the server.

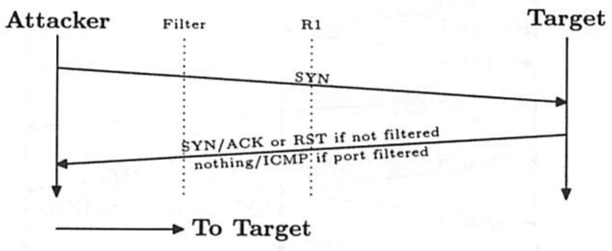


Figure 3: A simple port-scan. The attacker sends packets directly to the target to determine which ports are open.

set of attacks than the simple flooding explored in earlier work.

We view these attacks within the environment shown in Figure 2. We first present several probing attacks that can quickly determine a valid lightweight authenticator to defeat the DDoS protection. We then examine more sophisticated flooding attacks, and examine the effects of eavesdropping and compromise attacks. We assume that attackers may learn the ISP router topology by various means [28, 3] because it is a shared resource.

4.1 Probing

Several lightweight authenticators, such as destination port or destination address, allow arbitrary hosts to communicate directly with the target. While this provides flexibility and higher performance, it can be vulnerable to simple port-scanning attacks (Figure 3). If the target machine will reply to any packet with the lightweight authenticator, it is a trivial matter to scan, say, the 64,000 possible destination ports, or the 256 addresses in a /24 netblock. On a 100 Mbps Ethernet, a full port scan takes about 11 seconds. To prevent these attacks from succeeding, a **secondary key**, drawn from a large keyspace, is

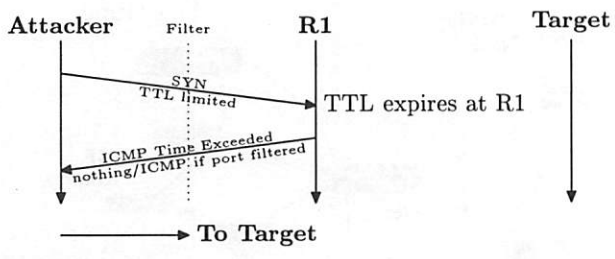


Figure 4: Firewalking. The attacker uses a traceroute-like mechanism to probe the ports that are allowed through the filter ring, without needing replies from the actual target.

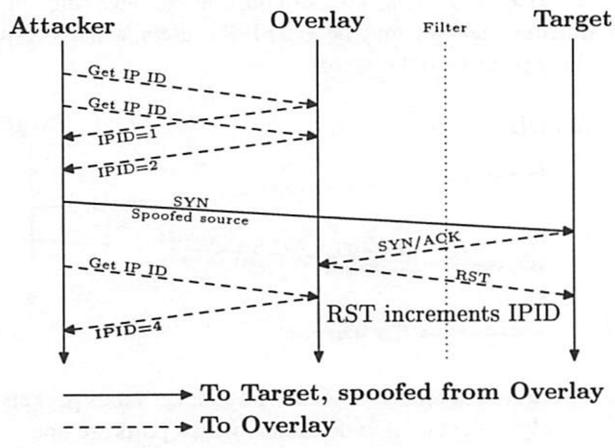


Figure 5: Idlescan indirect probing. The attacker spoofs a TCP SYN packet to the target. If the packet gets through the filter, the target replies with a TCP ACK to the overlay node. The overlay generates a RST because the connection does not exist. The attacker notices the IP ID increment at the overlay node when it sends the RST to determine if the packet got through the filter ring.

needed. While packets with a valid lightweight authenticator will go through the firewall, the server will respond to only packets with the proper secondary key. Clearly, this approach requires considerable attention to detail at the host level for filtering out host responses (ICMP port unreachables or TCP resets). The secondary key could be the addresses of the valid correspondent hosts, a key inside the packets, or heavier weight mechanisms such as IPSec.

The application of the secondary key is complicated by techniques such as *Firewalking* [14] that use Time-To-Live (TTL) tricks to determine what ports a firewall allows through, without requiring that the target host reply to such messages. Figure 4 shows an example of firewalking. Firewalking can be defeated by blocking ICMP TTL exceeded messages at the filter ring, but this breaks utilities like `traceroute` that rely on these messages.

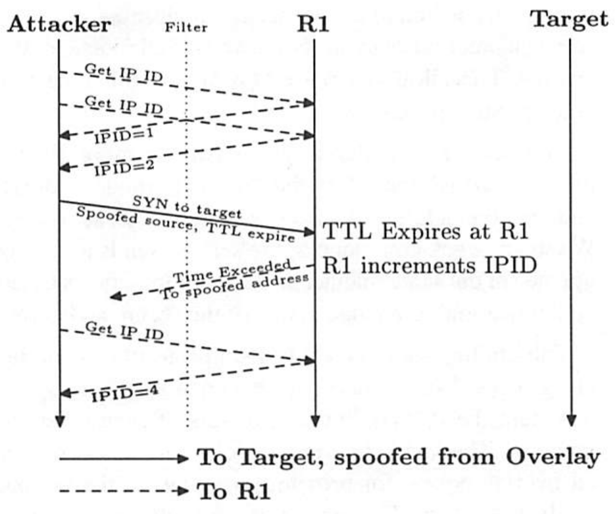


Figure 6: Next-hop scan. This attack combines the idlescan and firewalking to determine from an interior router if packets got through the firewall.

If the filter ring uses source address authentication, attackers can use indirect probing mechanisms to determine the set of source hosts that can reach the server. Tools such as Nmap [30] and Hping [22] can use IP ID increment scanning (or *Idlescanning*) [21] to scan a host indirectly via a third party. Figure 5 shows an idlescan wherein the attacker watches to see if the overlay node has received TCP ACK packets from the target. If it has, it will reply with a TCP RST packet, because it didn't originate a connection to the target. Transmitting this RST causes the overlay node to increment its IP ID, and therefore an attacker can conclude that the probe packet passed the filter by watching the overlay node's IP ID sequences. This technique is limited to TCP, and can be deterred by implementing IP ID randomization techniques on the overlay nodes. This technique also depends on low or predictable traffic volumes on the overlay nodes.

A variant on idlescanning that we call *next-hop* scanning can use other routers behind the filter ring to determine if spoofed packets are getting through. Figure 6 shows next-hop scanning. Like firewalking, next-hop scanning sends a TTL-limited probe at the target, which expires at some interior router $R1$. $R1$ generates an ICMP time exceeded message. Instead of directly receiving this message (if it's filtered or the source address was spoofed), the attacker indirectly observes the generation of the ICMP reply by the IP ID increment at $R1$.

Figure 7 shows a next-hop scan in action. Between sequence 93 and 94, the attacker machine sent 40 TTL-limited probes at the target, causing a large jump in $R1$'s IP ID. This trace was taken using `hping` on a production Cisco router on the Internet; the IP addresses have been

Source	Seq #	**IP ID change**	rtt
192.168.3.1	seq=91	id=+19	76.5 ms
192.168.3.1	seq=92	id=+16	233.4 ms
192.168.3.1	seq=93	id=+14	259.6 ms
192.168.3.1	seq=94	**id=+61**	76.2 ms
192.168.3.1	seq=95	id=+12	76.6 ms
192.168.3.1	seq=96	id=+10	75.5 ms

Figure 7: Next-hop scan showing IP ID increase at the router after the filter. After packet 93, the attacker sent a burst of packets that went through the filter. This scan method can be used to determine if spoofed packets are permitted to go towards a target, but it requires that the attacker be able to communicate with a router on the path after the filter.

obscured.

Other host vulnerabilities can be used in a similar way, but require that the overlay nodes run vulnerable software. Unlike application or specific host vulnerabilities, the IP ID scans are applicable to a wide array of host and router operating systems. They are difficult to defeat in an overlay context because they require either upgrades to the interior routers to prevent next-hop scanning from working, or much more extensive firewalling techniques than may be practical on shared core routers.

4.2 Timing Attacks

In an N-indirect Mayday network in which only certain overlay nodes are allowed to pass traffic to the server, a malicious client may be able to determine the identity of these nodes by timing analysis. Requests sent to an egress overlay node will often process more quickly than requests that must bounce through an extended series of intermediate nodes; in SOS, overlay traversal adds up to a factor of 10 increase in latency. This attack could allow an attacker to determine the identity of the egress node even in a randomly routed overlay. This attack can be mitigated by using multiple egress nodes and always relaying requests to a different egress node.

4.3 Adaptive flooding

This attack is one step up from blindly flooding the target with spoofed IPs. If the attacker can measure the response time of the target, by collusion with a legitimate client or passively monitoring clients, he can launch a more effective attack than pure flooding. The success of a DoS attack is not binary—intermediate levels of attack may simply slow down or otherwise impair the service provided.

Consider a lightweight authenticator whose keyspace has N possible values (all 64,000 TCP ports, or the 1,000 source addresses of overlay nodes). One value allows traffic to reach the target and consume limited resources. The target has a certain unused capacity, its *reserve*, R. The attacker can generate a certain amount of traffic, T. If $T > R$, the attacker uses up the target's resources, and the target's service degrades.

In most DDoS attacks, $T >> R$: the attacker's force is overwhelmingly large. In this case, the attacker can attack with *multiple* authenticators concurrently. If the attacker uses $\frac{N}{2}$ different authenticators, then 50% of the time, one of those authenticators is valid, and $\frac{2T}{N}$ traffic will reach the target. If the service slows down, the attacker knows that the authenticator was in the tested half of the keyspace. By recursively eliminating half of the remaining nodes in a binary-search like progression, the attacker can identify the authenticator in $O(\log N)$ attack rounds. After this, the full ferocity of the attack will penetrate the filter ring. Even intermediate rounds will likely damage the target.

This attack is slowed down by a large keyspace. When the attack power is sufficiently diluted (i.e., $\frac{2T}{N} < R$), the attack must first linearly probe small batches of the keyspace before identifying a range into which the binary search can proceed. Because this attack takes multiple rounds, key agility is effective at reducing the threat by allowing the system to change the key, ensuring that it remains fresh in the face of an attack.

4.4 Request Flood Attacks

Without careful attention to design, the overlay itself can be used by a malicious client to attack the target. An attacker can use the Akamai network, for example, by requesting identical content from many Akamai nodes concurrently. By reading very slowly from the network (or using an extremely small TCP receiver window), the attacker uses very little bandwidth. The caching overlay nodes, however, request the content as quickly as possible from the origin server, causing an overload.

These attacks are fairly easy to trace, and apply more to large, open systems (such as Akamai) than to closed systems with more trusted clients. However, they point out the need for caution when designing a system to improve performance or security, to ensure that the resulting nodes cannot themselves be used to launder or magnify a DoS attack.

4.5 Compromised Overlay Nodes

Controlling an overlay node allows an attacker not only the ability to see source/destination addresses, but to see the actual contents of the information flowing across the network. An attacker knows anything a compromised node knows.

Furthermore, the attacker can now launch internal attacks against the overlay itself. For example, the SOS system uses Chord [29] to perform routing lookups. The Chord system, and similar distributed hash tables, are themselves subject to a variety of attacks [25]. Any other component of the lookup system is similarly a potential source of cascaded compromise when an overlay node is compromised. This observation argues for keeping the overlay routing as simple as possible, unless the complexity results in needed security gains.

Proximity routing and singly-indirect routing can be immediately subverted when nodes are compromised. Doubly-indirect routing provides a degree of resilience to an attacker who compromises a node in a non-repeatable fashion (physical access, local misconfiguration, etc.). Random routing and mix routing can provide increased protection against compromise, but even these techniques will only delay an attacker who exploits a common flaw on the overlay nodes.

4.6 Identifying Attackers

It is possible to reverse the adaptive flooding attack to locate a single compromised node, if the lightweight authenticator can be changed easily. The search operates in an analogous fashion to the adaptive flooding attack: The server distributes key A to half of the nodes, and key B to the other half. When an attack is initiated with key A, the server knows that the attacker has compromised a machine in that half of the nodes. The search can then continue to narrow down the possibly compromised nodes until corrective action can be taken. This response almost certainly requires the agility of destination address authentication.

5 Analysis

Analysis of "backscatter" traffic suggests that more than 30% of observed DDoS SYN-flood or direct ICMP attacks involved 1000 packets per second (pps) or more, and that about 5% of them involved more than 10,000 pps [19]. This study did not observe indirect attacks that can take advantage of traffic amplifiers, and which can achieve even larger attack rates. Fortunately, these indirect attacks can often be stopped using source address authentication: There are no known attacks that can indirectly generate spoofed traffic.

How powerful are these attacks relative to the sites they attack? A T1 line (~ 1.54 Mbps) is likely the smallest access link that would be used by a "critical" service. With full-size packets (typically 1500 bytes), a T1 line can handle just 128 packets per second. The 30th percentile of DoS attacks is nearly an order of magnitude larger than this. A server in a co-location center with a 10 Mbps Ethernet connection can handle about 830 pps, and a 100 Mbps connected server could not withstand the upper 5% of DoS attacks at 10,000 pps.

For a victim on a T1 line, the top 5% of attacks could mount an adaptive flooding attack against a 100 node overlay with source authentication in under 8 rounds: Dividing the 10,000 pps by 50 nodes gives 200 packets per spoofed node per second, more than the T1 can handle. Thus, an attacker can immediately binary search in the egress node space, taking about $\log_2(100)$ rounds.

Many of the IP ID attacks take about 10 packets per attempted key. At 1000 pps, an attacker could discover a destination-port key in about five minutes. In a doubly-indirect overlay using source address authentication (SOS), the attacker could expect to locate the egress node's IP address in about 50 seconds. Using both of these keys, however, would force the attacker to spend nearly 4 days scanning at extremely high packet rates.

Resource consumption attacks, such as SYN floods, can be more destructive at lower packet rates; One study noted that a Linux webserver could handle only up to 500 pps of SYN packets before experiencing performance degradation [9]. SYN packets are also smaller, and are thus easier for an attacker to generate in large volume. By attacking multiple ingress nodes, and attacker could attempt to degrade the availability of the overlay. The top 5% of the attacks, over 10,000 pps, could disable about $\frac{10,000}{500} = 20$ overlay nodes. Modern TCP stacks with SYN cookies or compressed TCP state can handle higher packet rates than older systems, but SYN floods still consume more server resources than pure flooding attacks do.

6 Practical Deployment Issues

Could a Mayday system be practically deployed? We believe so. Service providers like Akamai [31] have existing overlay networks that number in the thousands of nodes. Over the last year, router vendors have created products like Juniper's M-series Internet Processor II ASIC that are capable of performing packet filtering at line speed on high-bandwidth links [17]. ISPs have historically been willing to implement filtering to mitigate extremely large DoS attacks; this willingness was tempered by the inability of their routers to do line-speed filtering. With the deployment of ASIC-assisted filters, ISPs should be able to deploy a few access list entries for major clients.

Mayday is primarily useful for protecting centralized services. Services may use a central server to ease their design, or it may not be economically feasible for a single service to purchase many under-utilized nodes to protect itself from attacks. In these cases, it may be particularly useful to take a service-provider approach, in which multiple clients contract with a single Mayday provider, who provides a shared overlay infrastructure. The service-provider approach helps amortize the cost of the overlay across multiple clients, and provides shared excess capacity to deal with transient load spikes. Protecting clients in this manner allows a larger overlay network, and reduces the number of entities that ISPs must deal with for creating router access lists.

Finally, DDoS protection is only the first line of defense for servers. The objective of Mayday is to prevent flooding attacks from overwhelming servers. In the real world, servers have a host of additional security problems that they must contend with, and interior lines of defense must still be maintained.

7 Conclusion and Future Work

We have presented a general architecture for using efficient router filtering with semi-trusted overlay nodes to provide denial of service resistance to servers. By generalizing from earlier work, we present several novel mechanisms that can provide improved performance with equivalent security. Designers implementing the Mayday architecture gain the flexibility to trade security for performance to better create a system that matches their needs.

To understand how overlay-based DoS protection would work in the real world, we presented several attacks that are effective against many router-based DoS prevention schemes. By providing options for more precise filtering and more agile rule updates, the Mayday architecture can successfully reduce the impact of these attacks.

While the Mayday architecture can provide a practical and effective proactive defense against DoS attacks, much work remains. Current router architectures are vulnerable to probes like our next-hop scan, and correcting these vulnerabilities will take time. Not all services can afford to protect themselves with Mayday, but still require some protection. There have been many proposals for detecting and preventing DoS attacks at the network layer and up, and sorting through the options remains a formidable task.

Acknowledgments

Many thanks to Alex Snoeren for suggesting the title, and being the initial sounding board for many of these ideas. Angelos Keromytis and Dan Rubenstein pointed out several improvements to this paper, and wrote the original SOS paper that inspired this work. Sanjit Biswas suggested the caching overlay node attack, and Mike Freedman helped shape the discussion about Tarzan and cover traffic. I am indebted to Hari Balakrishnan, Frans Kaashoek, Kevin Fu, Robert Morris, and the USITS reviewers for great discussion and feedback.

References

[1] Mazu networks. `http://www.mazunetworks.com/solutions/`, 2002.

[2] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, M., AND MORRIS, R. Resilient Overlay Networks. In *Proc. 18th ACM SOSP* (Banff, Canada, Oct. 2001), pp. 131–145.

[3] ANDERSEN, D. G., FEAMSTER, N., BAUER, S., AND BALAKRISHNAN, H. Topology inference from BGP routing dynamics. In *Proc. Internet Measurement Workshop* (Marseille, France, 2002).

[4] ARBOR NETWORKS. Peakflow for enterprises datasheet. `http://arbornetworks.com/up_media/up_files/Pflow_Enter_datasheet2.1.pdf%`, 2002.

[5] ASTA NETWORKS. Convergence of security and network performance (vantage system overview). `http://www.astanetworks.com/products/data_sheets/asta_data_sheet.pdf`, 2002.

[6] BELLOVIN, S. *ICMP Traceback Messages, Internet-Draft, draft-bellovin-itrace-00.txt, Work in Progress*, Mar. 2000.

[7] CAIDA Analysis of Code-Red. `http://www.caida.org/analysis/security/code-red/`, 2002.

[8] CHAUM, D. Untraceable electronic mail, return addresses and digital pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–88.

[9] DARMOHRAY, T., AND OLIVER, R. Hot spares for DoS attacks. *;Login: The Magazine of USENIX and SAGE* (July 2000).

[10] DEAN, D., FRANKLIN, M., AND STUBBLEFIELD, A. An algebraic approach to IP traceback. *Information and System Security 5*, 2 (2002), 119–137.

[11] FERGUSON, P., AND SENIE, D. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. RFC 2267, Jan. 1998.

[12] FERGUSON, P., AND SENIE, D. *Network Ingress Filtering*. Internet Engineering Task Force, May 2000. Best Current Practice 38, RFC 2827.

[13] FREEDMAN, M. J., AND MORRIS, R. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (Washington, D.C., Nov. 2002).

[14] GOLDSMITH, D., AND SCHIFFMAN, M. Firewalking: A traceroute-like analysis of IP packet responses to determine gateway access control lists. `http://www.packetfactory.net/firewalk/`, 1998.

[15] IOANNIDIS, J., AND BELLOVIN, S. M. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proc. Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2002).

[16] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE JR., J. W. Overcast: Reliable multicasting with an overlay network. In *Proc. 4th USENIX OSDI* (San Diego, California, October 2000), pp. 197–212.

[17] JUNIPER NETWORKS. M-series Internet Processor II ASIC Frequently Asked Questions. `http://www.juniper.net/solutions/faqs/m-series_ip2.html`.

[18] KEROMYTIS, A. D., MISRA, V., AND RUBENSTEIN, D. SOS: Secure overlay services. In *Proc. ACM SIGCOMM* (Pittsburgh, PA, 2002), pp. 61–72.

[19] MOORE, D., VOELKER, G., AND SAVAGE, S. Inferring Internet denial of service activity. In *Proc. USENIX Security Symposium* (Aug. 2001).

[20] PARK, K., AND LEE, H. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law Internets. In *Proc. ACM SIGCOMM* (San Diego, CA, 2001).

[21] SANFILIPPO, S. Posting about the IP ID reverse scan. `http://www.kyuzz.org/antirez/papers/dumbscan.html`, 1998.

[22] SANFILIPPO, S. hping home page. `http://www.hping.org/`, 2000.

[23] SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. The End-to-End Effects of Internet Path Selection. In *Proc. ACM SIGCOMM* (Boston, MA, 1999), pp. 289–299.

[24] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Network support for IP traceback. *IEEE/ACM Transactions on Networking 9*, 3 (June 2001).

[25] SIT, E., AND MORRIS, R. Security considerations for peer-to-peer distributed hash tables. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)* (Cambridge, MA, Feb. 2002).

[26] SNOEREN, A. C., AND BALAKRISHNAN, H. An End-to-End Approach to Host Mobility. In *Proc. 6th ACM/IEEE MOBICOM* (Aug. 2000).

[27] SNOEREN, A. C., PARTRIDGE, C., SANCHEZ, L. A., JONES, C. E., TCHAKOUNTIO, F., SCHWARTZ, B., KENT, S. T., AND STRAYER, W. T. Single-packet IP traceback. *IEEE/ACM Transactions on Networking (ToN) (to appear) 10*, 6 (Dec. 2002).

[28] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM* (Aug. 2002).

[29] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference* (San Diego, California, Aug. 2001).

[30] VASKOVICH, F. Nmap stealth port scanner. `http://www.insecure.org/nmap/index.html`, 2002.

[31] Akamai. http://www.akamai.com, 1999.

Adaptive Overload Control for Busy Internet Servers

Matt Welsh and David Culler
Intel Research, Berkeley
and University of California, Berkeley
{mdw,culler}@cs.berkeley.edu

Abstract

As Internet services become more popular and pervasive, a critical problem that arises is managing the performance of services under extreme overload. This paper presents a set of techniques for managing overload in complex, dynamic Internet services. These techniques are based on an adaptive admission control mechanism that attempts to bound the 90th-percentile response time of requests flowing through the service. This is accomplished by internally monitoring the performance of the service, which is decomposed into a set of event-driven *stages* connected with request queues. By controlling the rate at which each stage admits requests, the service can perform focused overload management, for example, by filtering only those requests that lead to resource bottlenecks. We present two extensions of this basic controller that provide class-based service differentiation as well as application-specific service degradation. We evaluate these mechanisms using a complex Web-based e-mail service that is subjected to a realistic user load, as well as a simpler Web server benchmark.

1 Introduction

Internet services have become a vital resource for many people. Internet-based e-mail, stock trading, driving directions, and even movie listings are often considered indispensable both for businesses and personal productivity. Web application hosting has opened up new demands for service performance and availability, with businesses relying on remotely-hosted services for accounting, human resources management, and other applications.

At the same time, Internet services are increasing in complexity and scale. Although much prior research has addressed the performance and scalability concerns of serving static Web pages [8, 28, 31], many modern services rely on dynamically-generated content, which requires significant amounts of computation and I/O to generate. It is not uncommon for a single Internet service request to involve several databases, application servers, and front-end Web servers. Unlike static content, dynamically-generated pages often cannot be cached or replicated for better performance, and the resource requirements for a given user load are very difficult to predict.

Moreover, Internet services are subject to enormous variations in demand, which in extreme cases can lead to *overload*. During overload conditions, the service's response times may grow to unacceptable levels, and exhaustion of resources may cause the service to behave erratically or even crash. The events of September 11, 2001 provided a poignant reminder of the inability of most Internet services to scale: many news sites worldwide were unavailable for several hours due to unprecedented demand. CNN.com experienced a request load 20 times greater than the expected *peak*, at one point exceeding 30,000 requests a second. Despite growing the size of the server farm, CNN was unable to handle the majority of requests to the site for almost 3 hours [23]. Many services rely on overprovisioning of server resources to handle spikes in demand. However, when a site is seriously overloaded, request rates can be orders of magnitude greater than the average, and it is clearly infeasible to overprovision a service to handle a 100-fold or 1000-fold increase in load.

Overload management is a critical requirement for Internet services, yet few services are designed with overload in mind. Often, services rely on the underlying operating system to manage resources, yet the OS typically does not have enough information about the service's resource requirements to effectively handle overload conditions. A common approach to overload control is to apply fixed (administrator-specified) resource limits, such as bounding the number of simultaneous socket connections or threads. However, it is difficult to determine the ideal resource limits under widely fluctuating loads; setting limits too low underutilizes resources, while setting them too high can lead to overload regardless. In addition, such resource limits do not have a direct relationship to client-perceived service performance.

We argue that Internet services should be designed from the ground up to detect and respond intelligently to overload conditions. In this paper, we present an architecture for Internet service design that makes overload management explicit in the programming model, providing services with the ability to perform fine-grained control of resources in response to heavy load. In this model, based on the *staged event-driven architecture* (SEDA) [40], services are constructed as a network of event-driven *stages* connected with explicit request *queues*. By applying admission control to each queue, the flow of requests through the service can be controlled in a focused manner.

To achieve scalability and fault-tolerance, Internet services are typically replicated across a set of machines, which may be within a single data center or geographically distributed [16, 37]. Even if a service is scaled across many machines, individual nodes still experience huge variations in demand. This requires that effective overload control techniques be deployed at the per-node level, which is the focus of this paper.

Our previous work on SEDA [40] addressed the efficiency and scalability of the architecture, and an earlier position paper [39] made the case for overload management primitives in service design. This paper builds on this work by presenting an adaptive admission control mechanism within the SEDA framework that attempts to meet a 90th percentile response time target by filtering requests at each stage of a service. This mechanism is general enough to support class-based prioritization of requests (e.g., allowing certain users to obtain better service than others) as well as application-specific service degradation.

Several prior approaches to overload control in Internet services have been proposed, which we discuss in detail in Section 5. Many of these techniques rely on static resource limits [3, 36], apply only to simplistic, static Web page loads [2, 9], or have been studied only under simulation [10, 20]. In contrast, the techniques described in this paper allow services to adapt to changing loads, apply to complex, dynamic Internet services with widely varying resource demands, and have been implemented in a realistic application setting. We evaluate our overload control mechanisms using both a resource-intensive Web-based e-mail service and a simple Web server benchmark. Our results show that these adaptive overload control mechanisms are effective at controlling the response times of complex Internet services, and permit flexible prioritization and degradation policies to be implemented.

2 The Staged Event-Driven Architecture

Our overload management techniques are based on the *staged event-driven architecture* (or SEDA), a model for designing Internet services that are inherently scalable and robust to load. In SEDA, applications are structured as a graph of event-driven *stages* connected with explicit *event queues*, as shown in Figure 1. We provide a brief overview of the architecture here; a more complete description and extensive performance results are given in [40].

2.1 SEDA Overview

SEDA is intended to support the massive concurrency demands of large-scale Internet services, as well as to exhibit good behavior under heavy load. Traditional server designs rely on processes or threads to capture the concurrency needs of the server: a common design is to devote a thread to each client connection. However, general-purpose threads are unable to scale to the large numbers required by busy Internet services [5, 17, 31].

The alternative to multithreading is event-driven concurrency, in which a small number of threads are used to process many simultaneous requests. However, event-driven server designs can often be very complex, requiring careful application-specific scheduling of request processing and I/O. This model also requires that application code never block, which is often difficult to achieve in practice. For example, garbage collection, page faults, or calls to legacy code can cause the application to block, leading to greatly reduced performance.

To counter the complexity of the standard event-driven approach, SEDA decomposes a service into a graph of *stages*, where each stage is an event-driven service component that performs some aspect of request processing. Each stage contains a small, dynamically-sized *thread pool* to drive its execution. Threads act as implicit continuations, automatically capturing the execution state across blocking operations; to avoid overusing threads, it is important that blocking operations be short or infrequent. SEDA provides nonblocking I/O primitives to eliminate the most common sources of long blocking operations.

Stages are connected with explicit *queues* that act as the execution boundary between stages, as well as a mechanism for controlling the flow of requests through the service. This design greatly reduces the complexity of managing concurrency, as each stage is responsible only for a subset of request processing, and stages are isolated from each other through composition with queues.

As shown in Figure 2, a stage consists of an *event handler*, an *incoming event queue*, and a dynamically-sized *thread pool*. Threads within a stage operate by pulling a *batch* of events off of the incoming event queue and invoking the application-supplied event handler. The event handler processes each batch of events, and dispatches zero or more events by enqueueing them on the event queues of other stages. The stage's incoming event queue is guarded by an *admission controller* that accepts or rejects new requests for the stage. The overload control mechanisms described in this paper are based on adaptive admission control for each stage in a SEDA service.

Additionally, each stage is subject to dynamic *resource control*, which attempts to keep each stage within its ideal operating regime by tuning parameters of the stage's operation. For example, one such controller adjusts the number of threads executing within each stage based on an observation of the stage's offered load (incoming queue length) and performance (throughput). This approach frees the application programmer from

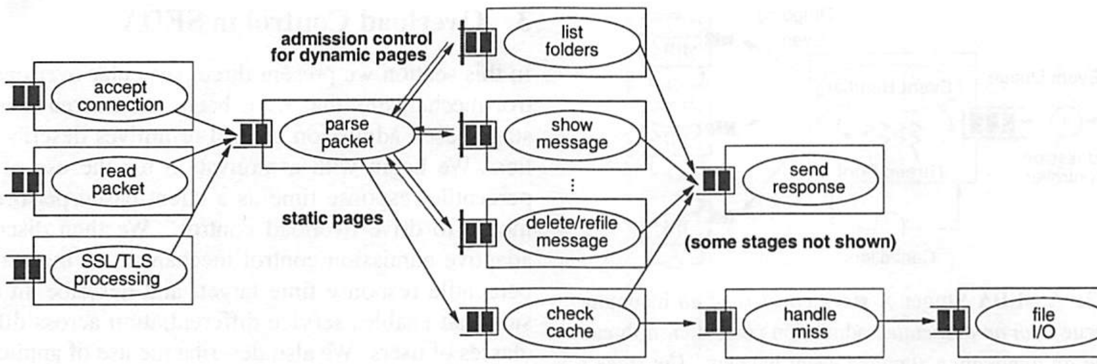


Figure 1: **Structure of the Arashi SEDA-based email service:** *This is a typical example of a SEDA-based Internet service, consisting of a network of stages connected with explicit event queues. Each stage is subject to adaptive resource management and admission control to prevent overload. Requests are read from the network and parsed by a the* read packet *and* parse packet *stages on the left. Each request is then passed to a stage that handles the particular request type, such as listing the user's mail folders. Static page requests are handled by a separate set of stages that maintain an in-memory cache. For simplicity, some event paths and stages have been elided from this figure.*

manually setting "knobs" that can have a serious impact on performance. More details on resource control in SEDA are given in [40].

2.2 Advantages of SEDA

While conceptually simple, the SEDA model has a number of desirable properties for overload management:

Exposure of the request stream: Event queues make the request stream within the service explicit, allowing the application (and the underlying runtime environment) to observe and control the performance of the system, e.g., through reordering or filtering of requests.

Focused, application-specific admission control: By applying fine-grained admission control to each stage, the system can avoid bottlenecks in a focused manner. For example, a stage that consumes many resources can be conditioned to load by throttling the rate at which events are admitted to just that stage, rather than refusing all new requests in a generic fashion. The application can provide its own admission control algorithms that are tailored for the particular service.

Modularity and performance isolation: Requiring stages to communicate through explicit event-passing allows each stage to be insulated from others in the system for purposes of code modularity and performance isolation.

2.3 Overload exposure and admission control

The goal of overload management is to prevent service performance from degrading in an uncontrolled fashion under heavy load, as a result of overcommitting resources. As a service approaches saturation, response times typically grow very large and throughput may degrade substantially. Under such conditions, it is often desirable to shed load, for example, by sending explicit rejection messages to users, rather than cause all users to experience unacceptable response times. Note that rejecting requests is just one form of load shedding; several alternatives are discussed below.

Overload protection in SEDA is accomplished through the use of fine-grained admission control at each stage, which can be used to implement a wide range of policies. Generally, by applying admission control, the service can limit the rate at which a stage accepts new requests, allowing performance bottlenecks to be isolated. A simple admission control policy might be to apply a fixed threshold to each stage's event queue; however, with this policy it is very difficult to determine what the ideal thresholds should be to meet some performance target. A better approach is for stages to monitor their performance and trigger rejection of incoming events when some performance threshold has been exceeded. Additionally, an admission controller could assign a cost to each event in the system, prioritizing low-cost events (e.g., inexpensive static Web page requests) over high-cost events (e.g., expensive dynamic pages). SEDA allows the admission control policy to be tailored for each individual stage.

This mechanism allows overload control to be performed within a service in response to measured resource bottlenecks; this is in contrast to "external" service control based on an *a priori* model of service capacity [2, 10]. Moreover, by performing admission control on a per-stage basis, overload response can be focused on those requests that lead to a bottleneck, and be customized for each type of request. This is as opposed to generic overload response mechanisms that fail to consider the nature of the request being processed [18, 19].

When the admission controller rejects a request, the corresponding enqueue operation fails, indicating to the

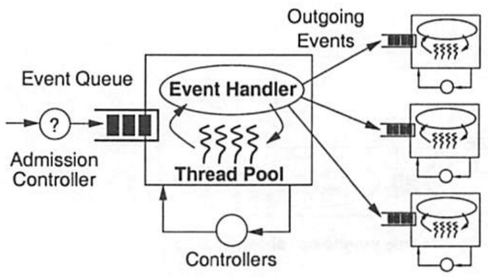


Figure 2: **A SEDA Stage:** *A stage consists of an* incoming event queue *with an associated* admission controller, *a* thread pool, *and an application-supplied* event handler. *The stage's operation is managed by a set of* controllers, *which dynamically adjust resource allocations and scheduling. The* admission controller *determines whether a given request is admitted to the queue.*

originating stage that there is a bottleneck in the system. The upstream stage is therefore responsible for reacting to these "overload signals" in some way. This explicit indication of overload differs from traditional service designs that treat overload as an exceptional case for which applications are given little indication or control.

Rejection of an event from a queue does not imply that the user's request is rejected from the system. Rather, it is the responsibility of the stage receiving a queue rejection to perform some alternate action, which depends greatly on the service logic, as described above. If the service has been replicated across multiple servers, the request can be redirected to another node, either internally or by sending an HTTP redirect message to the client. Services may also provide differentiated service by delaying certain requests in favor of others: an e-commerce site might give priority to requests from users about to complete an order. Another overload response is to block until the downstream stage can accept the request. This leads to backpressure, since blocked threads in a stage cause its incoming queue to fill, triggering overload response upstream. In some applications, however, backpressure may be undesirable as it causes requests to queue up, possibly for long periods of time.

More generally, an application may *degrade service* in response to overload, allowing a larger number of requests to be processed at lower quality. Examples include delivering lower-fidelity content (e.g., reduced-resolution image files) or performing alternate actions that consume fewer resources per request. Whether or not such degradation is feasible depends greatly on the nature of the service. The SEDA framework itself is agnostic as to the precise degradation mechanism employed—it simply provides the admission control primitive to signal overload to applications.

3 Overload Control in SEDA

In this section we present three particular overload control mechanisms that have been constructed using the stage-based admission control primitives described earlier. We begin with a motivation for the use of 90th-percentile response time as a client-based performance metric to drive overload control. We then discuss an adaptive admission control mechanism to meet a 90th-percentile response time target, and describe an extension that enables service differentiation across different classes of users. We also describe the use of application-specific service degradation in this framework.

3.1 Performance metrics

A variety of performance metrics have been studied in the context of overload management, including throughput and response time targets [9, 10], CPU utilization [2, 11, 12], and differentiated service metrics, such as the fraction of users in each class that meet a given performance target [20, 25]. In this paper, we focus on *90th-percentile response time* as a realistic and intuitive measure of client-perceived system performance. This metric has the benefit that it is both easy to reason about and captures the user's experience of Internet service performance. This is as opposed to average or maximum response time (which fail to represent the "shape" of a response time curve), or throughput (which depends greatly on the network connection to the service and bears little relation to user-perceived performance).

In this context, the system administrator specifies a target value for the service's 90th-percentile response time. The target response time may be parameterized by relative utility of the requests, for example, based on request type or user classification. An example might be to specify a lower response time target for requests from users with more items in their shopping cart. Our current implementation, discussed below, allows separate response time targets to be specified for each stage in the service, as well as for different classes of users (based on IP address, request header information, or HTTP cookies).

3.2 Response time controller design

The design of the per-stage overload controller in SEDA is shown in Figure 3. The controller consists of several components. A *monitor* measures response times for each request passing through a stage. Requests are tagged with the current time when they enter the service. At each stage S, the request's response time is calculated as the time it leaves S minus the time it entered the system. While this approach does not measure network effects, we expect that under overload the greatest contributor to perceived request latency will be intra-service

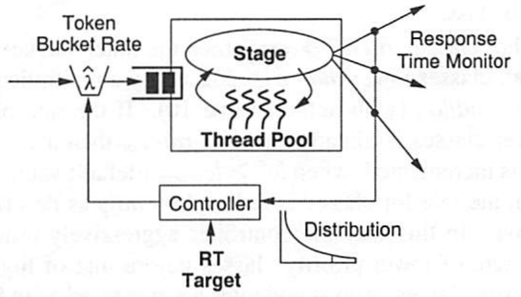


Figure 3: **Response time controller design:** *The controller observes a history of response times through the stage, and adjusts the rate at which the stage accepts new requests to meet an administrator-specified 90th-percentile response time target.*

Parameter	Description	Default value
nreq	# reqs before controller run	100
timeout	Time before controller run	1 sec
α	EWMA filter constant	0.7
err_i	% error to trigger increase	-0.5
err_d	% error to trigger decrease	0.0
adj_i	Additive rate increase	2.0
adj_d	Multiplicative rate decrease	1.2
c_i	Weight on additive increase	-0.1
$rate_{min}$	Minimum rate	0.05
ratemax	Maximum rate	5000.0

Figure 4: **Parameters used in the response time controller.**

response time.[1]

The measured 90th-percentile response time over some interval is passed to the *controller* that adjusts the *admission control parameters* based on the administrator-supplied response-time *target*. In the current design, the controller adjusts the rate at which new requests are admitted into the stage's queue by adjusting the rate at which new tokens are generated in a token bucket traffic shaper [33]. A wide range of alternate admission control policies are possible, including drop-tail FIFO or variants of random early detection (RED) [14].

The basic overload control algorithm makes use of additive-increase/multiplicative-decrease tuning of the token bucket rate based on the current observation of the 90th-percentile response time. The controller is invoked by the stage's event-processing thread after some number of requests (*nreq*) has been processed. The controller also runs after a set interval (*timeout*) to allow the rate to be adjusted when the processing rate is low.

The controller records up to *nreq* response-time samples and calculates the 90th-percentile sample *samp* by sorting the samples and taking the $(0.9 \times nreq)$-th sample.

[1]To avoid TCP's exponential backoff for initial SYN retransmission, our implementation of SEDA rapidly accepts new client connections. In cases where the number of incoming connections is extremely large, or initial SYNs are dropped by the network, our approach can be supplemented with some form of network latency estimation [30, 35] to obtain a more accurate response-time estimate.

In order to prevent sudden spikes in the response time sample from causing large reactions in the controller, the 90th-percentile response time estimate is smoothed using an exponentially weighted moving average with parameter α:

$$cur = \alpha \cdot cur + (1 - \alpha) \cdot samp$$

The controller then calculates the error between the current response time measurement and the target:

$$err = \frac{cur - target}{target}$$

If $err > err_d$, the token bucket rate is reduced by a multiplicative factor adj_d. If $err < err_i$, the rate is increased by an additive factor proportional to the error: $-(err - c_i)adj_i$. The constant c_i is used to weight the rate increase such that when $err = c_i$ the rate adjustment is 0.

The parameters used in the implementation are summarized in Figure 4. These parameters were determined experimentally using a combination of microbenchmarks with artificial loads and real applications with realistic loads (e.g., the e-mail service described in the next section). In most cases the controller algorithm and parameters were tuned by running test loads against a service and observing the behavior of the controller in terms of measured response times and the corresponding admission rate.

These parameters have been observed to work well across a range of applications, however, there are no guarantees that they are optimal. In particular, the behavior of the controller is sensitive to the setting of the smoothing filter constant α, as well as the rate increase adj_i and decrease adj_d; the setting of the other parameters is less critical. The main goal of tuning is allow the controller to react quickly to increases in response time, while not being so conservative that an excessive number of requests are rejected. An important problem for future investigation is the tuning (perhaps automated) of controller parameters in this environment. It would be useful to apply concepts from control theory to aid in the tuning process, but this requires the development of complex models of system behavior. We discuss the role of control theoretic techniques in more detail in Section 5.3.

3.3 Class-based differentiation

By prioritizing requests from certain users over others, a SEDA application can implement various policies related to class-based service level agreements (SLAs). A common example is to prioritize requests from "gold" customers, who might pay more money for the privilege, or to give better service to customers with items in their shopping cart.

Various approaches to class-based differentiation are possible in SEDA. One option would be to segregate request processing for each class into its own set of stages,

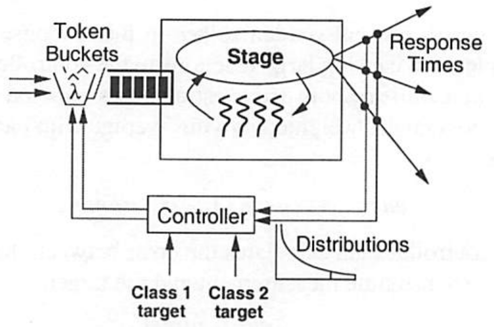
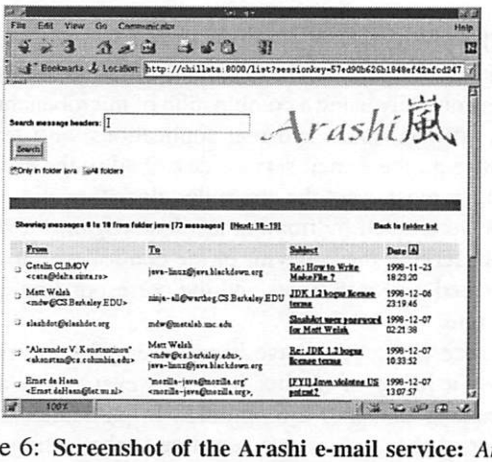


Figure 5: **Multiclass overload controller design:** *For each request class, the controller measures the 90th-percentile response time, and adjusts the rate at which the stage accepts new requests of each class. When overload is detected, the admission rate for lower-priority classes is reduced before that of higher-priority classes.*


File Edit View Go Communicator Help
Bookmarks Location: http://chillata:8000/list?sessionkey=57ed90b626b1848ef42afcd247
Search message headers:
Arashi 嵐
Search
Only in folder java
All folders
100%

Showing messages 8 to 18 from folder java [73 messages] [Next: 18-19]   Back to folder list

From	To	Subject	Date
Catalin CLIMOV <cata@delta.sinta.ro>	java-linux@java.blackdown.org	Re: How to Write MakeFile ?	1998-11-25 18:23:20
Matt Welsh <mdw@CS.Berkeley.EDU>	ninja-all@warthog.CS.Berkeley.EDU	JDK 1.2 bogus license terms	1998-12-06 23:19:46
slashdot@slashdot.org	mdw@metalab.unc.edu	Slashdot user password for Matt Welsh	1998-12-07 02:21:38
"Alexander V. Konstantinou" <akonstan@cs.columbia.edu>	Matt Welsh <mdw@cs.berkeley.edu>, java-linux@java.blackdown.org	Re: JDK 1.2 bogus license terms	1998-12-07 10:33:52
Ernst de Haan <Ernst.deHaan@let.uu.nl>	"mozilla-java@mozilla.org" <mozilla-java@mozilla.org>,	[FYI] Java violates US patent?	1998-12-07 13:07:57

Figure 6: **Screenshot of the Arashi e-mail service:** *Arashi allows users to read e-mail through a Web browser interface. Many traditional e-mail reader features are implemented, including message search, folder view, sorting message lists by author, subject, or date fields, and so forth.*

in effect partitioning the service's stage graph into separate flows for each class. In this way, stages for higher-priority classes could be given higher priority, e.g., by increasing scheduling priorities or allocating additional threads. Another option is to process all classes of requests in the same set of stages, but make the admission control mechanism aware of each class, for example, by rejecting a larger fraction of lower-class requests than higher-class requests. This is the approach taken here.

The multiclass response time control algorithm is identical to that presented in Section 3.2, with several small modifications. Incoming requests are assigned an integer *class* that is derived from application-specific properties of the request, such as IP address or HTTP cookie information. A separate instance of the response time controller is used for each class c, with independent response time targets $target^c$. Likewise, the queue admission controller maintains a separate token bucket for each class.

For class c, if $err^c > err^c_d$, then the token bucket rate of all classes *lower than* c is reduced by a multiplicative factor $adjlo_d$ (with default value 10). If the rate of all lower classes is already equal to $rate_{min}$ then a counter lc^c is incremented; when $lc^c \geq lc_{thresh}$ (default value 20), then the rate for class c is reduced by adj_d as described above. In this way the controller aggressively reduces the rate of lower-priority classes before that of higher-priority classes. Admission rates are increased as in Section 3.2, except that whenever a higher-priority class exceeds its response time target, all lower-priority classes are flagged to prevent their admission rates from being increased during the next iteration of the controller.

3.4 Service degradation

Another approach to overload management is to allow applications to degrade the quality of delivered service in order to admit a larger number of requests [1, 7, 16]. SEDA itself does not implement service degradation mechanisms, but rather signals overload to applications in a way that allows them to degrade if possible. Stages can obtain the current 90th-percentile response time measurement as well as enable or disable the stage's admission control mechanism. This allows an service to implement degradation by periodically sampling the current response time and comparing it to the target. If service degradation is ineffective (say, because the load is too high to support even the lowest quality setting), the stage can re-enable admission control to cause requests to be rejected.

4 Evaluation

In this section, we evaluate the SEDA overload control mechanisms using two applications: a complex Web-based e-mail service, and a Web server benchmark involving dynamic page generation that is capable of degrading service in response to overload.

4.1 Arashi: A SEDA-based e-mail service

We wish to study the behavior of the SEDA overload controllers in a highly dynamic environment, under a wide variation of user load and resource demands. We have developed the *Arashi*[2] Web-based e-mail service, which is akin to Hotmail and Yahoo! Mail, allowing users to access e-mail through a Web browser interface with various functions: managing e-mail folders, deleting and refiling messages, searching for messages, and so forth. A screenshot of the Arashi service is shown in Figure 6.

Arashi is implemented using the *Sandstorm* platform, a SEDA-based Internet services framework implemented

[2]*Arashi* is the Japanese word for *storm*.

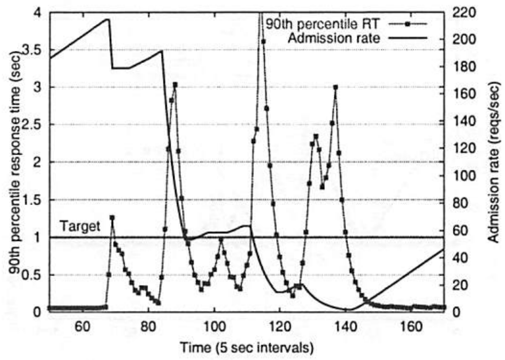


Figure 7: **Overload controller operation:** *This figure shows the operation of the SEDA overload controller for one of the stages in the Arashi e-mail service during a large load spike. A load spike of 1000 users enters the system at around $t = 70$ and leaves the system around $t = 150$. The response time target is set to 1 sec. The overload controller responds to a spike in response time by exponentially decreasing the admission rate of the stage. Likewise, when the measured response time is below the target, the admission rate is increased slowly. Notice the slight increase in the admission rate around $t = 100$; this is an example of the proportional increase of the admission rate based on the error between the response time measurement and the target. The spikes in the measured response time are caused by bursts of requests entering the stage, as well as resource contention across stages.*

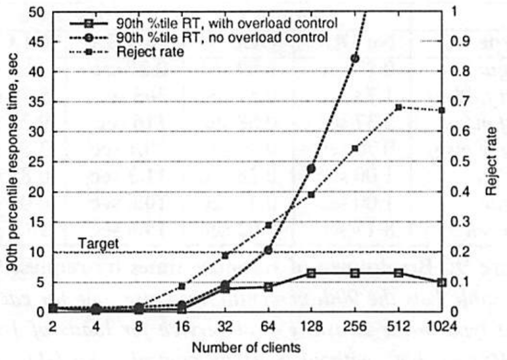


Figure 8: **Overload control in Arashi:** *This figure shows the 90th-percentile response time for the Arashi e-mail service with and without the overload controller enabled. The 90th-percentile response time target is 10 sec. Also shown is the fraction of rejected requests with overload control enabled. Note that the overload controller is operating independently on each request type, though this figure shows the 90th-percentile response time and reject rate averaged across all requests. As the figure shows, the overload control mechanism is effective at meeting the response time target despite a many-fold increase in load.*

in Java [40].[3] As shown in Figure 1, some number of stages are devoted to generic Web page processing, nonblocking network and file I/O, and maintaining a cache of recently-accessed static pages; in the Arashi service, there is only a single static Web object (the Arashi logo image). Arashi employs six stages to process dynamic page requests, with one stage assigned to each request type (show message, list folders, etc.). Each stage is implemented as a Java class that processes the corresponding request type, accesses e-mail data from a MySQL [27] database, and generates a customized HTML page in response. This design allows the admission control parameters for each request type to be tuned independently, which is desirable given the large variation in resource requirements across requests. For example, displaying a single message is a relatively lightweight operation, while listing the contents of an entire folder requires a significant number of database accesses.

The client load generator used in our experiments emulates a number of simultaneous users, each accessing a single mail folder consisting of between 1 and 12794 messages, the contents of which are based on actual e-mail archives. Emulated users access the service based on a simple Markovian model of user behavior derived from traces of the UC Berkeley Computer Science Division's IMAP server.[4] The inter-request think time is aggressively set to 20ms. When the service rejects a request from a user, the user waits for 5 sec before attempting to log into the service again. The Arashi service runs on a 1.2 GHz Pentium 4 machine running Linux 2.4.18, and the client load generators run on between 1 and 16 similar machines connected to the server with Gigabit Ethernet. Since we are only interested in the overload behavior of the server, WAN network effects are not incorporated into our evaluation.

4.2 Controller operation

Figure 7 demonstrates the operation of the overload controller, showing the 90th-percentile response time measurement and token bucket admission rate for one of the stages in the Arashi service (in this case, for the "list folders" request type). Here, the stage is being subjected to a very heavy load spike of 1000 users, causing response times to increase dramatically.

As the figure shows, the controller responds to a spike in the response time by exponentially decreasing the token bucket rate. When the response time is below the target, the rate is increased slowly. Despite several overshoots of the response time target, the controller is very effective at keeping the response time near the target. The response time spikes are explained by two factors. First, the request load is extremely bursty due to the realistic nature of the client load generator. Second, because

[3]Our earlier work [40] demonstrated that despite using Java, Sandstorm performance is competitive with systems implemented in C.

[4]We are indebted to Steve Czerwinski for providing the IMAP trace data.

	16 users		*1024 users*	
Type	**No OLC**	**OLC**	**No OLC**	**OLC**
login	0.83 sec	0.59 sec	0.86 sec	3.84 sec
list folders	1.73 sec	0.57 sec	**365 sec**	5.75 sec
list msgs	2.37 sec	0.58 sec	**116 sec**	9.28 sec
show msg	0.70 sec	0.30 sec	**30.1 sec**	3.87 sec
delete	1.00 sec	0.28 sec	**11.3 sec**	6.85 sec
refile	1.00 sec	0.47 sec	**10.6 sec**	6.07 sec
search	8.17 sec	9.92 sec	**19.6 sec**	**18.1 sec**

Figure 9: **Breakdown of response times by request type:** *This table lists the 90th-percentile response time for each request type in the Arashi e-mail service for loads of 16 and 1024 users, both without overload control ("No OLC") and with overload control ("With OLC"). The response time target is 10 sec, and values in boldface exceeded the target. Although request types exhibit a widely varying degree of complexity, the controller is effective at meeting the response time target for each type. With 1024 users, the performance target is exceeded for* search *requests due to their relative infrequency.*

all stages share the same back-end database, requests for other stages (not shown in the figure) may cause resource contention that affects the response time of the "list folders" stage. Note, however, that the largest response time spike is only about 4 seconds, which is not too serious given a response time target of 1 second. With no admission control, response times grow without bound, as we will show in Sections 4.3 and 4.4.

4.3 Overload control with increased user load

Figure 8 shows the 90th-percentile response time of the Arashi service, as a function of the user load, both with and without the per-stage overload controller enabled. Also shown is the fraction of overall requests that are rejected by the overload controller. The 90th-percentile response time target is set to 10 sec. For each data point, the corresponding number of simulated clients load the system for about 15 minutes, and response-time distributions are collected after an initial warm-up period of about 20 seconds. As the figure shows, the overload control mechanism is effective at meeting the response time target despite a many-fold load increase.

Recall that the overload controller is operating on each request type separately, though this figure shows the 90th-percentile response time and reject rate across *all* requests. Figure 9 breaks the response times down according to request type, showing that the overload controller is able to meet the performance target for each request type individually. With 1024 users, the performance target is exceeded for *search* requests. This is mainly due to their relative infrequency: search requests are very uncommon, comprising less than 1% of the request load. The controller for the *search* stage is therefore unable to react as quickly to arrivals of this request type.

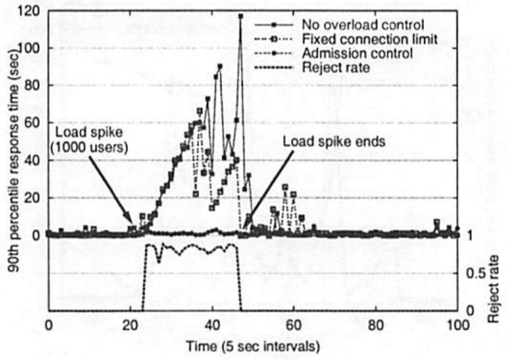


Figure 10: **Overload control under a massive load spike:** *This figure shows the 90th-percentile response time experienced by clients using the Arashi e-mail service under a massive load spike (from 3 users to 1000 users). Without overload control, response times grow without bound; with overload control (using a 90th-percentile response time target of 1 second), there is a small increase during load but response times quickly stabilize. The lower portion of the figure shows the fraction of requests rejected by the overload controller.*

4.4 Overload control under a massive load spike

The previous section evaluated the overload controller under a steadily increasing user load, representing a slow increase in user population over time. We are also interested in evaluating the effectiveness of the overload controller under a sudden load spike. In this scenario, we start with a base load of 3 users accessing the Arashi service, and suddenly increase the load to 1000 users. This is meant to model a "flash crowd" in which a large number of users access the service all at once.

Figure 10 shows the performance of the overload controller in this situation. Without overload control, there is an enormous increase in response times during the load spike, making the service effectively unusable for all users. With overload control and a 90th-percentile response time target of 1 second, about 70-80% of requests are rejected during the spike, but response times for admitted requests are kept very low.

Our feedback-driven approach to overload control is in contrast to the common technique of limiting the number of client TCP connections to the service, which does not actively monitor response times (a small number of clients could cause a large response time spike), nor give users any indication of overload. In fact, refusing TCP connections has a negative impact on user-perceived response time, as the client's TCP stack transparently retries connection requests with exponential backoff. Figure 10 shows the client response times when overload control is disabled and a limit of 128 simultaneous connections is imposed on the server. As the figure shows, this approach leads to large response times overall. During this benchmark run, over 561 of the 1000 clients ex-

	Per-stage AC		*Single-stage AC*	
Type	**90th RT**	**Rejected**	**90th RT**	**Rejected**
login	2.07 sec	44.3%	1.00 sec	18.8%
list folders	8.11 sec	59.6%	3.97 sec	59.6%
list msgs	8.04 sec	47.1%	6.20 sec	53.7%
show msg	3.90 sec	23.1%	2.04 sec	49.1%
delete	4.86 sec	11.3%	3.26 sec	51.4%
refile	4.60 sec	10.4%	2.12 sec	54.7%
search	**22.2 sec**	0%	**18.9 sec**	53.3%

Figure 11: **Comparison of per-stage versus single-stage admission control:** *This table shows the 90th-percentile response time and reject rate by request type for a load of 128 users on the Arashi service. The response time target is 10 sec, and times shown in boldface exceeded the performance target. With per-stage admission control, the rejection rate is tuned based on the overhead of each request type. For single-stage admission control, all requests experience approproximately the same rejection rate.*

perienced connection timeout errors.

We claim that giving 20% of the users good service and 80% of the users some indication that the site is overloaded is better than giving *all* users unacceptable service. However, this comes down to a question of what policy a service wants to adopt for managing heavy load. Recall that the service need not reject requests outright—it could redirect them to another server, degrade service, or perform an alternate action. The SEDA design allows a wide range of policies to be implemented: in the next section we look at degrading service as an alternate response to overload.

Applying admission control to each stage in the Arashi service allows the admission rate to be separately tuned for each type of request. An alternative policy would be to use a single admission controller that filters all incoming requests, regardless of type. Under such a policy, a small number of expensive requests can cause the admission controller to reject many unrelated requests from the system. Figure 11 compares these two policies, showing the admission rate and 90th-percentile response time by request type for a load of 128 users. As the figure shows, using a single admission controller is much more aggressive in terms of the overall rejection rate, leading to lower response times overall. However, the policy does not discriminate between infrequent, expensive requests and more common, less expensive requests.

4.5 Service degradation experiments

As discussed previously, SEDA applications can respond to overload by degrading the fidelity of the service offered to clients. This technique can be combined with admission control, for example, by rejecting requests only when the lowest service quality still leads to overload.

We evaluate the use of service degradation through a simple Web server benchmark that incorporates a con-

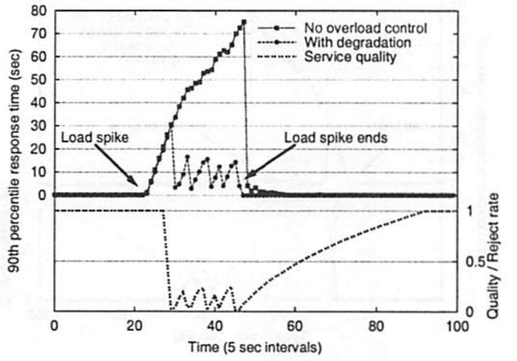


Figure 12: **Effect of service degradation:** *This figure shows the 90th-percentile response time experienced by clients accessing a simple service consisting of a single bottleneck stage. The stage is capable of degrading the quality of service delivered to clients in order to meet response time demands. The 90th-percentile response time target is set to 5 seconds. Without service degradation, response times grow very large under a load spike of 1000 users. With service degradation, response times are greatly reduced, oscillating near the target performance level.*

tinuous "quality knob" that can be tuned to trade performance for service fidelity. A single stage acts as a CPU-bound bottleneck in this service; for each request, the stage reads a varying amount of data from a file, computes checksums on the file data, and produces a dynamically generated HTML page in response. The stage has an associated quality factor that controls the amount of data read from the file and the number of checksums computed. By reducing the quality factor, the stage consumes fewer CPU resources, but provides "lower quality" service to clients.

Using the overload control interfaces in SEDA, the stage monitors its own 90th-percentile response time and reduces the quality factor when it is over the administrator-specified limit. Likewise, the quality factor is increased slowly when the response time is below the limit. Service degradation may be performed either independently or in conjunction with the response-time admission controller described above. If degradation is used alone, then under overload all clients are given service but at a reduced quality level. In extreme cases, however, the lowest quality setting may still lead to very large response times. The stage optionally re-enables the admission controller when the quality factor is at its lowest setting and response times continue to exceed the target.

Figure 12 shows the effect of service degradation under an extreme load spike, and Figure 13 shows the use of service degradation coupled with admission control. As these figures show, service degradation alone does a fair job of managing overload, though re-enabling the admission controller under heavy load is much more effective.

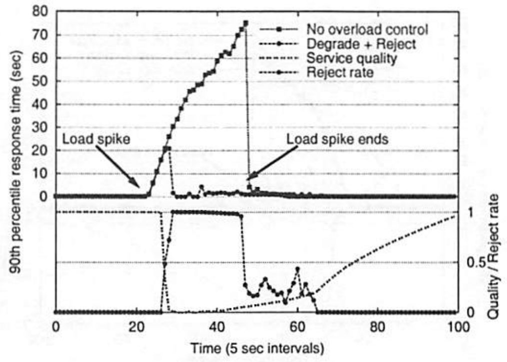


Figure 13: **Service degradation combined with admission control:** *This figure shows the effect of service degradation combined with admission control. The experiment is identical to that in Figure 12, except that the bottleneck stage re-enables admission control when the service quality is at its lowest level. In contrast to the use of service degradation alone, degradation coupled with admission control is much more effective at meeting the response time target.*

Note that when admission control is used, a very large fraction (99%) of the requests are rejected; this is due to the extreme nature of the load spike and the inability of the bottleneck stage to meet the performance target, even at a degraded level of service.

4.6 Service differentiation

Finally, we evaluate the use of multiclass service differentiation, in which requests from lower-priority users are rejected before those from higher-priority users. In these experiments, we deal with two user classes, each with a 90th-percentile response time target of 10 sec, generating load against the Arashi e-mail service. Each experiment begins with a base load of 128 users from the lower-priority class. At a certain point during the run, 128 users from the higher-priority class also start accessing the service, and leave after some time. The user class is determined by a field in the HTTP request header; the implementation is general enough to support class assignment based on client IP address, HTTP cookies, or other information.

Figure 14 shows the performance of the multiclass overload controller without service differentiation enabled: all users are effectively treated as belonging to the same class. As the figure shows, the controller is able to maintain response times near the target, though no preferential treatment is given to the high priority requests.

In Figure 15, service differentiation is enabled, causing requests from lower-priority users to be rejected more frequently than higher-priority users. As the figure demonstrates, while both user classes are active, the overall rejection rate for higher-priority users is slightly lower than that in Figure 14, though the lower-priority class is penalized with a higher rejection rate. The average reject rate is 87.9% for the low-priority requests, and 48.8% for the high-priority requests. This is compared to 55.5% and 57.6%, respectively, when no service differentiation is performed. Note that the initial load spike (around $t = 100$) when the high priority users become active is somewhat more severe with service differentiation enabled. This is because the controller is initially attempting to reject only low-priority requests, due to the lag threshold (lc_{thresh}) for triggering admission rate reduction for high-priority requests.

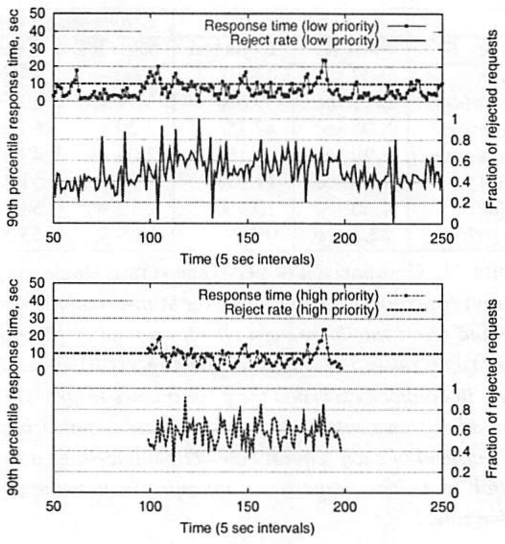


Figure 14: **Multiclass experiment without service differentiation:** *This figure shows the operation of the overload control mechanism in Arashi with two classes of 128 users each accessing the service. The high-priority users begin accessing the service at time* $t = 100$ *and leave at* $t = 200$. *No service differentiation is used, so all users are treated as belonging to the same class. The 90th-percentile response time target is set to 10 sec. The controller is able to maintain response times near the target, though no preferential treatment is given to higher-priority users as they exhibit an identical frequency of rejected requests.*

5 Related Work

In this section we survey prior work in Internet service overload control, discussing previous approaches as they relate to four broad categories: resource containment, admission control, control-theoretic approaches, and service degradation. In [38] we present a more thorough overview of related work.

5.1 Resource containment

The classic approach to resource management in Internet services is static resource containment, in which *a priori* resource limits are imposed on an application or service to avoid overcommitment. We categorize all of these approaches as *static* in the sense that some external

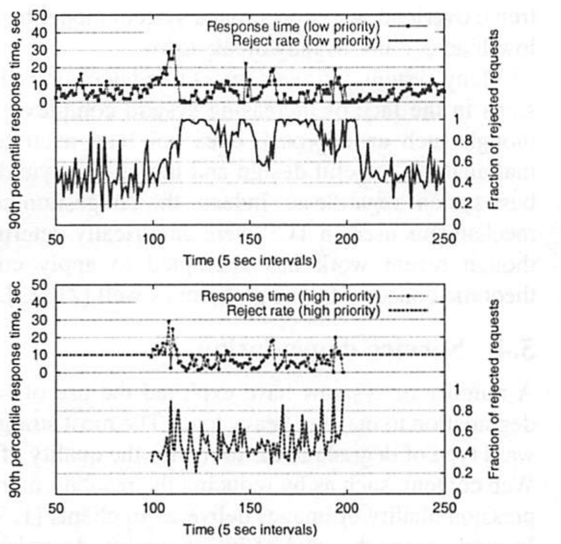


Figure 15: **Multiclass service differentiation:** *This figure shows the operation of the multiclass overload control mechanism in Arashi with two classes of 128 users each. Service differentiation between the two classes is enabled and the 90th-percentile response time target for each class is 10 sec. The high-priority users begin accessing the service at time* $t = 100$ *and leave at* $t = 200$. *As the figure shows, when the high-priority users become active, there is an initial load spike that is compensated for by penalizing the admission rate of the low-priority users. Overall the low-priority users receive a large number of rejections, while high-priority users are able to receive a greater fraction of service.*

entity (say, the system administrator) imposes a limit on the resource usage of a process or application. Although resource limits may change over time, they are typically not driven by monitoring and feedback of system performance; rather, the limits are arbitrary and rigid.

In a traditional thread-per-connection Web server design, the only overload mechanism generally used is to bound the number of processes (and hence the number of simultaneous connections) that the server will allocate. When all server threads are busy, the server stops accepting new connections; this is the type of overload protection used by Apache [4]. There are two serious problems with this approach. First, it is based on a static thread or connection limit, which does not directly correspond to user-perceived performance. Service response time depends on many factors such as user load, the length of time a given connection is active, and the type of request (e.g., static versus dynamic pages). Secondly, not accepting new TCP connections gives the user no indication that the site is overloaded: the Web browser simply reports that it is still waiting for a connection to the site. As described earlier, this wait time can grow to be very long, due to TCP's exponential backoff on SYN retransmissions.

Zeus [41] and thttpd [3] provide mechanisms to throttle the bandwidth consumption for certain Web pages to prevent overload, based on a static bandwidth limit imposed by the system administrator for certain classes of requests. A very similar mechanism has been described by Li and Jamin [24]. In this model, the server intentionally delays outgoing replies to maintain a bandwidth limit, which has the side-effect of tying up server resources for greater periods of time to deliver throttled replies.

5.2 Admission control

The use of admission control as an overload management technique has been explored by several systems. Many of the proposed techniques are based on fixed policies, such as bounding the maximum request rate of requests to some constant value. Another common aspect of these approaches is that they often reject incoming work to a service by refusing to accept new client TCP connections.

Iyer *et al.* [18] describe a simple admission control mechanism based on bounding the length of the Web server request queue. This work analyzes various settings for the queue abatement and discard thresholds, though does not specify how these thresholds should be set to meet any given performance target. Cherkasova and Phaal [11] present *session-based* admission control, driven by a CPU utiliization threshold, which performs an admission decision when a new session arrives from a user, rather than for individual requests or connections. Such a policy would be straightforward to implement in SEDA.

Voigt *et al.*[36] present several kernel-level mechanisms for overload management: restricting incoming connections based on dropping SYN packets; parsing and classification of HTTP requests in the kernel; and ordering the socket listen queue by request URL and client IP address. Another traffic-shaping approach is described in [19], which drives the selection of incoming packet rates based on an observation of system load, such as CPU utilization and memory usage. Web2K [6] brings several of these ideas together in a Web server "front-end" that performs admission control based on the length of the request queue; as in [18], the issue of determining appropriate queue thresholds is not addressed. Lazy Receiver Processing [13] prevents the TCP/IP receive path from overloading the system; this technique could be coupled with SEDA's overload controllers in extreme cases where incoming request rates are very high. Qie *et al.*[34] introduce resource limitations within the Flash [31] Web serber, primarily as a protection against denial-of-service attacks, though the idea could be extended to overload control.

Several other admission control mechanisms have been presented, though often only in simulation or for simplistic workloads (e.g., static Web pages). PAC-

ERS [10] attempts to limit the number of admitted requests based on expected server capacity, but this paper deals with a simplistic simulated service where request processing time is linear in the size of the requested Web page. A related paper allocates requests to Apache server processes to minimize per-class response time bounds [9]. This paper is unclear on implementation details, and the proposed technique silently drops requests if delay bounds are exceeded, rather than explicitly notifying clients of overload. Kanodia and Knightly [20] develop an admission control mechanism based on *service envelopes*, a modelling technique used to characterize the traffic of multiple flows over a shared link. The admission controller attempts to meet response-time bounds for multiple classes of service requests, but again is only studied under a simple simulation of Web server behavior.

5.3 Control-theoretic approaches

Control theory [29] provides a formal framework for reasoning about the behavior of dynamic systems and feedback-driven control. A number of control-theoretic approaches to performance management of real systems have been described [26, 32], and several of these have focused on overload control for Internet services.

Abdelzaher and Lu [2] describe an admission control scheme that attempts to maintain a CPU utilization target using a proportional-integral (PI) controller and a simplistic linear model of server performance. Apart from ignoring caching, resource contention, and a host of other effects, this model is limited to static Web page accesses. An alternative approach, described in [25], allocates server processes to each class of pending connections to obtain a *relative delay* between user classes. Diao *et al.* [12] describe a control-based mechanism for tuning Apache server parameters (the number of server processes and the per-connection idle timeout) to meet given CPU and memory utilization targets. Recall that in Apache, reducing the number of server processes leads to increased likelihood of stalling incoming connections; although this technique effectively protects server resources from oversaturation, it results in poor client-perceived performance.

Although control theory provides a useful set of tools for designing and reasoning about systems subject to feedback, there are many challenges that must be addressed in order for these techniques to be applicable to real-world systems. One of the greatest difficulties is that good models of system behavior are often difficult to derive. Unlike physical systems, which can often be described by linear models or approximations, Internet services are subject to poorly understood traffic and internal resource demands. The systems described here all make use of linear models, which may not be accurate in describing systems with widely varying loads and resource requirements. Moreover, when a system is subject to extreme overload, we expect that a system model based on low-load conditions may break down.

Many system designers resort to *ad hoc* controller designs in the face of increasing system complexity. Although such an approach does not lend itself to formal analysis, careful design and tuning may yield a robust system regardless. Indeed, the congestion-control mechanisms used in TCP were empirically determined, though recent work has attempted to apply control-theoretic concepts to this problem as well [21, 22].

5.4 Service degradation

A number of systems have explored the use of service degradation to manage heavy load. The most straightforward type of degradation is to reduce the quality of static Web content, such as by reducing the resolution or compression quality of images delivered to clients [1, 7, 16]. In many cases the goal of image quality degradation is to reduce network bandwidth consumption on the server, though this may have other effects as well, such as memory savings.

A more sophisticated example of service degradation involves replacing entire Web pages (with many inlined images and links to other expensive objects) with stripped-down Web pages that entail fewer individual HTTP requests to deliver. This was the approach taken by CNN.com on September 11, 2001; in response to overload, CNN replaced its front page with simple HTML page that that could be transmitted in a single Ethernet packet [23]. This technique was implemented manually, though a better approach would be to degrade service gracefully and automatically in response to load.

In some cases it is possible for a service to make performance tradeoffs in terms of the freshness, consistency, or completeness of data delivered to clients. Brewer and Fox [15] describe this tradeoff in terms of the *harvest* and *yield* of a data operation; harvest refers to the amount of data represented in a response, while yield (closely related to availability) refers to the probability of completing a request. For example, a Web search engine could reduce the amount of the Web database searched when overloaded, and still produce results that are good enough such that a user may not notice any difference.

One disadvantage to service degradation is that many services lack a "fidelity knob" by design. For example, an e-mail or chat service cannot practically degrade service in response to overload: "lower-quality" e-mail and chat have no meaning. In these cases, a service must resort to admission control, delaying responses, or one of the other mechanisms described earlier. Indeed, rejecting a request through admission control is the lowest quality setting for a degraded service.

6 Future Work and Conclusions

We have argued that measurement and control are the keys to resource management and overload protection in busy Internet services. This is in contrast to long-standing approaches based on resource containment, which typically mandate an *a priori* assignment of resources to each request, limiting the range of applicable load-conditioning policies. Still, introducing feedback as a mechanism for overload control raises a number of questions, such as how controller parameters should be tuned. We have relied mainly on a heuristic approach to controller design, though more formal, control-theoretic techniques are possible [32]. Capturing the performance and resource needs of real applications through detailed models is an important research direction if control-theoretic techniques are to be employed more widely.

The use of per-stage admission control allows the service to carefully control the flow of requests through the system, for example, by only rejecting those requests that lead to a bottleneck resource. The downside of this approach is that a request may be rejected late in the processing pipeline, after it has consumed significant resources in upstream stages. There are several ways to address this problem. Ideally, request classification and load shedding can be performed early; in Arashi, requests are classified in the first stage after they have been read from the network. Another approach is for a stage's overload controller to affect the admission-control policy of upstream stages, causing requests to be dropped before encountering a bottleneck. This paper has focused on stage-local reactions to overload, though a global approach is also feasible in the SEDA framework.

Our approach to overload management is based on adaptive admission control using "external" observations of stage performance. This approach uses no knowledge of the actual resource-consumption patterns of stages in an application, but is based on the implicit connection between request admission and performance. This does not directly capture all of the relevant factors that can drive a system into overload. For example, a memory-intensive stage (or a stage with a memory leak) can lead to VM thrashing even with a very low request-admission rate. One direction for future work is to inform the overload control mechanism with more direct measurements of per-stage resource consumption. We have investigated one step in this direction, a system-wide resource monitor capable of signaling stages when resource usage (e.g., memory availability or CPU utilization) meets certain conditions. In this model, stages receive system-wide overload signals and use the information to voluntarily reduce their resource consumption.

We have presented an approach to overload control for dynamic Internet services, based on adaptive, per-stage admission control. In this approach, the system actively observes application performance and tunes the admission rate of each stage to attempt to meet a 90th-percentile response time target. We have presented extensions of this approach that perform class-based service differentiation as well as application-specific service degradation. The evaluation of these control mechanisms, using both the complex Arashi e-mail service and a simpler dynamic Web server benchmark, show that they are effective for managing load with increasing user populations as well as under massive load spikes.

Software Availability

The SEDA software, related papers, and other documentation are available for download from `http://www.cs.berkeley.edu/~mdw/proj/seda`.

References

[1] T. F. Abdelzaher and N. Bhatti. Adaptive content delivery for Web server QoS. In *Proceedings of International Workshop on Quality of Service*, London, June 1999.

[2] T. F. Abdelzaher and C. Lu. Modeling and performance control of Internet servers. In *Invited Paper, 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[3] Acme Labs. thttpd: Tiny/Turbo/Throttling HTTP Server. `http://www.acme.com/software/thttpd/`.

[4] Apache Software Foundation. The Apache Web server. `http://www.apache.org`.

[5] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[6] P. Bhoj, S. Ramanathan, and S. Singhal. Web2K: Bringing QoS to Web Servers. Technical Report HPL-2000-61, HP Labs, May 2000.

[7] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated multimedia Web services using quality aware transcoding. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, January 1996.

[9] H. Chen and P. Mohapatra. Session-based overload control in QoS-aware Web servers. In *Proceedings of IEEE INFOCOM 2002*, New York, June 2002.

[10] X. Chen, H. Chen, and P. Mohapatra. An admission control scheme for predictable server response time for Web accesses. In *Proceedings of the 10th World Wide Web Conference*, Hong Kong, May 2001.

[11] L. Cherkasova and P. Phaal. Session based admission control: A mechanism for improving the performance of an overloaded Web server. Technical Report HPL-98-119, HP Labs, June 1998.

[12] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *Proceedings of the Network Operations and Management Symposium 2002*, Florence, Italy, April 2002.

[13] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[15] A. Fox and E. A. Brewer. Harvest, yield and scalable tolerant systems. In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

[16] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[17] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[18] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for Web servers. In *Workshop on Performance and QoS of Next Generation Networks*, Nagoya, Japan, November 2000.

[19] H. Jamjoom, J. Reumann, and K. G. Shin. QGuard: Protecting Internet servers from overload. Technical Report CSE-TR-427-00, University of Michigan Department of Computer Science and Engineering, 2000.

[20] V. Kanodia and E. Knightly. Multi-class latency-bounded Web services. In *Proceedings of IEEE/IFIP IWQoS 2000*, Pittsburgh, PA, June 2000.

[21] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high bandwidth-delay product environments. In *Proceedings of ACM SIGCOMM 2002*, Pittsburgh, PA, August 2002.

[22] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM 1991*, September 1991.

[23] W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA'01, December 2001.

[24] K. Li and S. Jamin. A measurement-based admission-controlled Web server. In *Proceedings of IEEE Infocom 2000*, Tel-Aviv, Israel, March 2000.

[25] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in Web servers. In *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, June 2001.

[26] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and evaluation of a feedback control EDF algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

[27] MySQL AB. MySQL. `http://www.mysql.com`.

[28] National Laboratory for Applied Network Research. The Squid Internet object cache. `http://www.squid-cache.org`.

[29] K. Ogata. *Modern Control Engineering*. Prentice Hall, 1997.

[30] D. P. Olshefski, J. Nieh, and D. Agrawal. Inferring client response time at the Web server. In *Proceedings of SIGMETRICS 2002*, Marina Del Rey, California, June 2002.

[31] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[32] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[33] C. Partridge. *Gigabit Networking*. Addison-Wesley, 1993.

[34] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[35] R. Rajamony and M. Elnozahy. Measuring client-perceived response times on the WWW. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[36] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, June 2001.

[37] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, June 2002.

[38] M. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*. PhD thesis, UC Berkeley, August 2002.

[39] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[40] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[41] Zeus Technology. Zeus Web Server. `http://www.zeus.co.uk/products/ws/`.

Model-Based Resource Provisioning in a Web Service Utility

Ronald P. Doyle*
IBM
Research Triangle Park
rdoyle@us.ibm.com

Jeffrey S. Chase
Omer M. Asad†
Wei Jin
Amin M. Vahdat
Department of Computer Science‡
Duke University
{chase, jin, vahdat}@cs.duke.edu

Abstract

Internet service utilities host multiple server applications on a shared server cluster. A key challenge for these systems is to provision shared resources on demand to meet service quality targets at least cost. This paper presents a new approach to utility resource management focusing on coordinated provisioning of memory and storage resources. Our approach is *model-based*: it incorporates internal models of service behavior to predict the value of candidate resource allotments under changing load. The model-based approach enables the system to achieve important resource management goals, including differentiated service quality, performance isolation, storage-aware caching, and proactive allocation of surplus resources to meet performance goals. Experimental results with a prototype demonstrate model-based dynamic provisioning under Web workloads with static content.

1 Introduction

The hardware resources available to a network service determine its maximum request throughput and—under typical network conditions—a large share of the response delays perceived by its clients. As hardware performance advances, emphasis is shifting from server software performance (e.g., [19, 26, 27, 37]) to improving the manageability and robustness of large-scale services [3, 5, 12, 31]. This paper focuses on a key subproblem: automated on-demand resource provisioning for multiple competing services hosted by a shared server infrastructure—a utility. It applies to Web-based services in a shared hosting center or a Content Distribution Network (CDN).

The utility allocates each service a *slice* of its resources, including shares of memory, CPU time, and available throughput from storage units. Slices provide performance isolation and enable the utility to use its resources efficiently. The slices are chosen to allow each hosted service to meet service quality targets (e.g., response time) negotiated in Service Level Agreements (SLAs) with the utility. Slices vary dynamically to respond to changes in load and resource status. This paper addresses the *provisioning* problem: how much resource does a service need to meet SLA targets at its projected load level? A closely related aspect of utility resource allocation is *assignment*: which servers and storage units will provide the resources to host each service?

Previous work addresses various aspects of utility resource management, including mechanisms to enforce resource shares (e.g., [7, 9, 36]), policies to provision shares adaptively [12, 21, 39], admission control with probabilistically safe overbooking [4, 6, 34], scheduling to meet SLA targets or maximize yield [21, 22, 23, 32], and utility data center architectures [5, 25, 30].

The key contribution of this paper is to demonstrate the potential of a new *model-based* approach to provisioning multiple resources that interact in complex ways. The premise of model-based resource provisioning (MBRP) is that internal models capturing service workload and behavior can enable the utility to predict the effects of changes to the workload intensity or resource allotment. Experimental results illustrate model-based dynamic provisioning of memory and storage shares for hosted Web services with static content. Given adequate models, this approach may generalize to a wide range of services including complex multi-tier services [29] with interacting components, or services with multiple functional stages [37]. Moreover, model-based provisioning is flexible enough to adjust to resource constraints or surpluses exposed during assignment.

This paper is organized as follows. Section 2 motivates the work and summarizes our approach. Section 3 out-

*R. Doyle is also a PhD student at Duke Computer Science.
†O. Asad is currently at Sun Microsystems.
‡This work is supported by the U.S. National Science Foundation through grants CCR-00-82912, ANI-126231, and EIA-9972879, and by IBM, Network Appliance, and HP Laboratories.

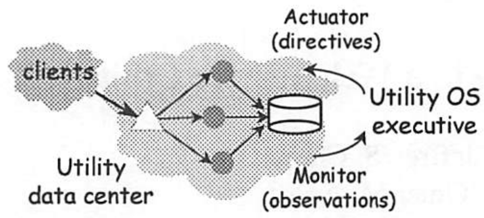


Figure 1: A utility OS. A feedback-controlled policy module or *executive* monitors load and resource status, determines resource slices for each service, and issues directives to configure server shares and direct request traffic to selected servers.

lines simple models for Web services; Section 4 describes a resource allocator based on the models, and demonstrates its behavior in various scenarios. Section 5 describes our prototype and presents experimental results. Section 6 sets our approach in context with related work, and Section 7 concludes.

2 Overview

Our utility provisioning scheme is designed to run within a feedback-controlled resource manager for a server cluster, as depicted in Figure 1. The software that controls the mapping of workloads to servers and storage units is a *utility operating system*; it supplements the operating systems on the individual servers by coordinating traditional OS functions (e.g. resource management and isolation) across the utility as a whole. An executive policy component continuously adjusts resource slices based on smoothed, filtered observations of traffic and resource conditions, as in our previous work on the Muse system [12]. The utility enforces slices by assigning complete servers to services, or by partitioning the resources of a physical server using a resource control mechanism such as resource containers [9] or a virtual machine monitor [36]. Reconfigurable redirecting switches direct request traffic toward the assigned servers for each service.

This paper deals with the executive's policies for provisioning resource slices in this framework. Several factors make it difficult to coordinate provisioning for multiple interacting resources:

- **Bottleneck behavior.** Changing the allotment of resources other than the primary bottleneck has little or no effect. On the other hand, changes at a bottleneck affect demand for other resources.
- **Global constraints.** Allocating resources under constraint is a zero-sum game. Even when data centers are well-provisioned, a utility OS must cope with constraints when they occur, e.g., due to abnormal load surges or failures. Memory is often constrained; in the general case there is not enough to cache the entire data set for every service. Memory management can dramatically affect service performance [26].
- **Local constraints.** Assigning service components (*capsules*) to specific nodes leads to a bin-packing problem [3, 34]. Solutions may expose local resource constraints or induce workload interference.
- **Caching.** Sizing of a memory cache in one component or stage can affect the load on other components. For example, the OS may overcome a local constraint at a shared storage unit by increasing server memory allotment for one capsule's I/O cache, freeing up storage throughput for use by another capsule.

The premise of MBRP is that network service loads have common properties that allow the utility OS to predict their behavior. In particular, service loads are streams of requests with stable average-case behavior; the model allows the system to adapt to changing resource demands at each stage by continuously feeding observed request arrival rates to the models to predict resource demands at each stage. Moreover, the models enable the system to account for resource interactions in a comprehensive way, by predicting the effects of planned resource allotments and placement choices. For example, the models can answer questions like: "how much memory is needed to reduce this service's storage access rate by 20%?". Figure 2 depicts the use of the models within the utility OS executive.

MBRP is a departure from traditional resource management using reactive heuristics with limited assumptions about application behavior. Our premise is that MBRP is appropriate for a utility OS because it hosts a smaller number of distinct applications that are both heavily resource-intensive and more predictable in their average per-request resource demands. In many cases the workloads and service behavior have been intensively studied. We propose to use the resulting models to enable dynamic, adaptive, automated resource management.

For example, we show that MBRP can provision for average-case response time targets for Web services with static content under dynamically varying load. First, the system uses the models to generate initial *candidate* resource allotments that it predicts will meet response time targets for each service. Next, an *assignment* phase maps service components to specific servers and storage units in the data center, balancing affinity, migration cost, competing workloads, and local constraints on individual servers and storage units. The system may

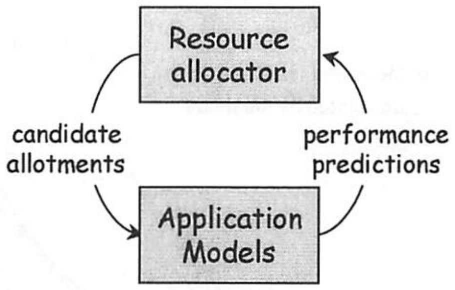


Figure 2: Using models to evaluate candidate resource allotments in a utility OS. The executive refines the allotments until it identifies a resource assignment that meets system goals.

adjust candidate allotments to compensate for local constraints or resource surpluses discovered during the assignment phase. In this context, MBRP meets the following goals in an elegant and natural way:

- **Differentiated service quality**. Since the system can predict the effects of candidate allotments on service quality, it can plan efficient allotments to meet response time targets, favoring services with higher value or more stringent SLAs.

- **Resource management for diverse workloads.** Each service has a distinct *profile* for reference locality and the CPU-intensity of requests. The model parameters capture these properties, enabling the executive to select allotments to meet performance goals. These may include allocating surplus resources to optimize global metrics (e.g., global average response time, yield, or profit).

- **Storage-aware caching**. Storage units vary in their performance due to differences in their configurations or assigned loads. Models capture these differences, enabling the resource scheduler to compensate by assigning different amounts of memory to reduce the dependence on storage where appropriate. Recent work [16] proposed reactive kernel heuristics with similar goals.

- **On-line capacity planning**. The models can determine aggregate resource requirements for all hosted services at observed or possible load levels. This is useful for admission control or to vary hosting capacity with offered load.

This flexibility in meeting diverse goals makes MBRP a powerful basis for proactive resource management in utilities.

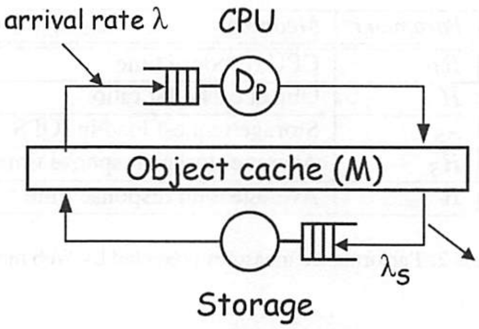


Figure 3: A simple model for serving static Web content. Requests arrive at rate λ (which varies with time) and incur an average service demand D_P in a CPU. An in-memory cache of size M absorbs a portion H of the requests as hits; the misses generate requests to storage at rate λ_S.

Parameter	*Meaning*
α	Zipf locality parameter
λ	Offered load in requests/s
S	Average object size
T	Total number of objects
M	Memory size for object cache
D_P	Average per-request CPU demand
μ_S, ϕ	Peak storage throughput in IOPS

Table 1: Input parameters for Web service models.

3 Web Service Models

This section presents simplified analytical models to predict performance of Web services with static content, based on the system model depicted in Figure 3. Table 1 summarizes the model parameters and other inputs. Table 2 summarizes the model outputs, which guide the choices of the resource allocator described in Section 4.

The models are derived from basic queuing theory and recent work on performance modeling for Web services. They focus on average-case behavior and abstract away much detail. For example, they assume that the network within the data center is well-provisioned. Each of the models could be extended to improve its accuracy; what is important is the illustration they provide of the potential for model-based resource provisioning.

3.1 Server Memory

Many studies indicate that requests to static Web objects follow a Zipf-like popularity distribution [11, 38]. The probability p_x of a request to the xth most popular object is proportional to $1/x^\alpha$, for some parameter α. Many requests target the most popular objects, but the distribution has a heavy tail of unpopular objects with poor reference locality. Higher α values increase the concentra-

Parameter	*Meaning*
R_P	CPU response time
H	Object cache hit ratio
λ_S	Storage request load in IOPS
R_S	Average storage response time
R	Average total response time

Table 2: Performance measures predicted by Web models.

tion of requests on the most popular objects. We assume that object size is independent of popularity [11], and that size distributions are stable for each service [35].

Given these assumptions, a utility OS can estimate hit ratio for a memory size M from two parameters: α, and T, the total size of the service's data (consider M and T to be in units of objects). If the server effectively caches the most popular objects (i.e., assuming perfect Least Frequently Used or LFU replacement), and ignoring object updates, the predicted object hit ratio H is given by the portion of the Zipf-distributed probability mass that is concentrated in the M most popular objects. We can closely approximate this H by integrating over a continuous form of the Zipf probability distribution function [11, 38]. The closed form solution reduces to:

$$H = \frac{1 - M^{1-\alpha}}{1 - T^{1-\alpha}} \qquad (1)$$

Zipf distributions appear to be common in a large number of settings, so this model is more generally applicable. While pure LFU replacement is difficult to implement, a large body of Web caching research has yielded replacement algorithms that approximate LFU; even poor schemes such as LRU are qualitatively similar in their behavior.

3.2 Storage I/O Rate

Given an estimate of cache hit ratio H, the service's storage I/O load is easily derived: for a Web load of λ requests per second, the I/O throughput demand λ_S for storage (in IOPS, or I/O operations per second) is:

$$\lambda_S = \lambda S(1 - H) \qquad (2)$$

The average object size S is given as the number of I/O operations to access an object on a cache miss.

3.3 Storage Response Time

To determine the performance impact of disk I/O, we must estimate the average response time R_S from storage. We model each storage server as a simple queuing

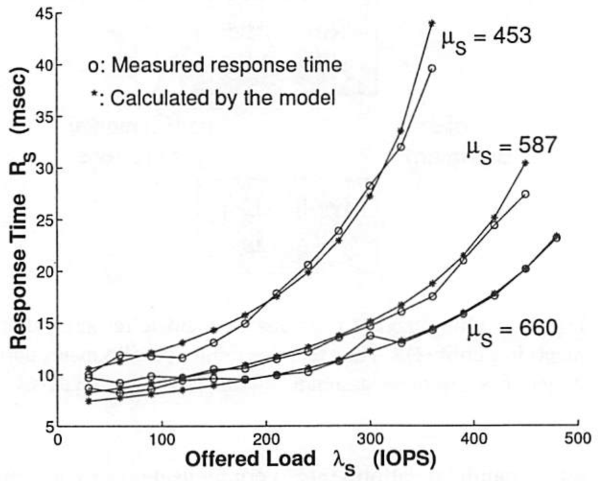


Figure 4: Predicted and observed R_S for an NFS server (Dell 4400, $k = 4$ Seagate Cheetah disks, 256MB, BSD/UFS) under three synthetic file access workloads with different fileset sizes.

center, parameterized for k disks; to simplify the presentation, we assume that the storage servers in the utility are physically homogeneous. Given a stable request mix, we estimate the utilization of a storage server at a given IOPS request load λ_S as λ_S/μ_S, where μ_S is its saturation throughput in IOPS for that service's file set and workload profile. We determine μ_S empirically for a given request mix. A well-known response time formula then predicts the average storage response time as:

$$R_S = \frac{k/\mu_S}{1 - (\lambda_S/\mu_S)} \qquad (3)$$

This model does not reflect variations in cache behavior within the storage server, e.g., from changes to M or λ_S. Given the heavy-tailed nature of Zipf distributions, Web server miss streams tend to show poor locality when the Web server cache (M) is reasonably provisioned [14].

Figure 4 shows that this model closely approximates storage server behavior at typical utilization levels and under low-locality synthetic file access loads (random reads) with three different fileset sizes (T). We used the FreeBSD Concatenated Disk Driver (CCD) to stripe the filesets across $k = 4$ disks, and the *fstress* tool [2] to generate the workloads and measure R_S.

Since storage servers may be shared, we extend the model to predict R_S when the service receives an allotment ϕ of storage server resources, representing the maximum storage throughput in IOPS available to the service within its share.

$$R_S = \frac{k/\phi}{1-(\lambda_S/\phi)} \quad (4)$$

This formula assumes that some scheduler enforces proportional shares ϕ/μ_S for storage. Our prototype uses *Request Windows* [18] to approximate proportional sharing. Other approaches to differentiated scheduling for storage [23, 33] are also applicable.

3.4 Service Response Time

We combine these models to predict average total response time R for the service, deliberately ignoring congestion on the network paths to the clients (which the utility OS cannot control). Given a measure D_P of average per-request service demand on the Web server CPU, CPU response time R_P is given by a simple queuing model similar to the storage model above; previous work [12] illustrates use of such a model to adaptively provision CPU resources for Web services. The service's average response time R is simply:

$$R = R_P + R_S(1-H) \quad (5)$$

This ignores the CPU cost of I/O and the effects of prefetching for large files. The CPU and storage models already account (crudely) for these factors in the average case.

3.5 Discussion

These models are cheap to evaluate and they capture the key behaviors that determine application performance. They were originally developed to improve understanding of service behavior and to aid in static design of server infrastructures; since they predict how resource demands change as a function of offered load, they can also act as a basis for dynamic provisioning in a shared hosting utility. A key limitation is that the models assume a stable average-case per-request behavior, and they predict only average-case performance. For example, the models here are not sufficient to provision for probabilistic performance guarantees. Also, since they do not account for interference among workloads using shared resources, MBRP depends on performance isolation mechanisms (e.g., [9, 36]) that limit this interference. Finally, the models do not capture overload pathologies [37]; MBRP must assign sufficient resources to avoid overload, or use dynamic admission control to prevent it.

Importantly, the models are independent of the MBRP framework itself, so it is possible to replace them with more powerful models or extend the approach to a wider range of services. For example, it is easy to model simple dynamic content services with a stable average-case service demand for CPU and memory. I/O patterns for database-driven services are more difficult to model, but a sufficient volume of requests will likely reflect a stable average-case behavior.

MBRP must parameterize the models for each service with the inputs from Table 1. T and S parameters and average-case service demands are readily obtainable (e.g., as in Muse [12] or Neptune [32]), but it is an open question how to obtain α and μ_S from dynamic observations. The system can detect anomalies by comparing observed behavior to the model predictions, but MBRP is "brittle" unless it can adapt or reparameterize the models when anomalies occur.

4 A Model-Based Allocator

This section outlines a resource provisioning algorithm that plans least-cost resource slices based on the models from Section 3. The utility OS executive periodically invokes it to adjust the allotments, e.g., based on filtered load and performance observations. The output is an *allotment vector* for each service, representing a CPU share together with memory and storage allotments $[M, \phi]$. The provisioning algorithm comprises three primitives designed to act in concert with an assignment planner, which maps the allotted shares onto specific servers and storage units within the utility. The resource provisioning primitives are as follows:

- *Candidate* plans initial candidate allotment vectors that it predicts will meet SLA response time targets for each service at its load λ.
- *LocalAdjust* modifies a candidate vector to adapt to a local resource constraint or surplus exposed during assignment. For example, the assignment planner might place some service on a server with insufficient memory to supply the candidate M; *LocalAdjust* constructs an alternative vector that meets the targets within the resource constraint, e.g., by increasing ϕ to reduce R_S.
- *GroupAdjust* modifies a set of candidate vectors to adapt to a resource constraint or surplus exposed during assignment. It determines how to reduce or augment the allotments to optimize a global metric, e.g., to minimize predicted average response time. For example, the assignment planner might assign multiple hosted services to share a network storage unit; if the unit is not powerful enough to meet the aggregate resource demand, then *GroupAdjust* modifies each vector to conform to the constraint.

We have implemented these primitives in a prototype executive for a utility OS. The following subsections discuss each of these primitives in turn.

4.1 Generating Initial Candidates

To avoid searching a complex space of resource combinations to achieve a given performance goal, *Candidate* follows a simple principle: *build a balanced system.* The allocator configures CPU and storage throughput (ϕ) allotments around a predetermined average utilization level ρ_{target}. The ρ_{target} may be set in a "sweet spot" range from 50-80% that balances efficient use of storage and CPU resources against queuing delays incurred at servers and storage units. The value for ρ_{target} is a separate policy choice. Lower values for ρ_{target} provide more "headroom" to absorb transient load bursts for the service, reducing the probability of violating SLA targets. The algorithm generalizes to independent ρ_{target} values for each $(service, resource)$ pair. ,

The *Candidate* algorithm consists of the following steps:

- **Step 1.** Predict CPU response time R_P at the configured ρ_{target}, as described in Section 3.4. Select initial $\phi = \mu_S$.
- **Step 2.** Using ϕ and ρ_{target}, predict storage response time R_S using Equation (4). Note that the allocator does not know λ_S at this stage, but it is not necessary because R_S depends only on ϕ and the ratio of λ_S/ϕ, which is given by ρ_{target}.
- **Step 3.** Determine the required server memory hit ratio (H) to reach the SLA target response time R, using Equation (5) and solving for H as:

$$H = 1 - \frac{R - R_P}{R_S} \qquad (6)$$

- **Step 4.** Determine the storage arrival rate λ_S from λ and H, using Equation (2). Determine and assign the resulting candidate storage throughput allotment $\phi = \lambda_S/\rho_{target}$.
- **Step 5.** Loop to step 2 to recompute storage response time R_S with the new value of ϕ. Loop until the difference of ϕ values is within a preconfigured percentage of μ_S.
- **Step 6.** Determine and assign the memory allotment M necessary to achieve the hit ratio H, using Equation (1).

Note that the candidate M is the minimum required to meet the response time target. Given reasonable targets, *Candidate* leaves memory undercommitted. To illustrate, Figure 5 shows candidate storage and memory allotments $[M, \phi]$ for an example service as offered Web load λ increases along the x-axis. *Candidate* responds to increasing load by increasing ϕ rather than M. This is because increasing λ does not require a higher hit ratio H to meet a fixed response time target. For a fixed M and corresponding H, λ_S grows linearly with λ, and so the storage allotment ϕ also grows linearly following $\lambda_S/\phi = \rho_{target}$.

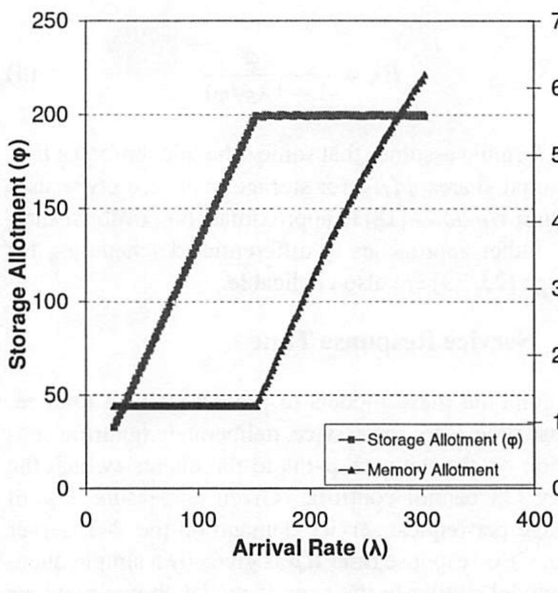
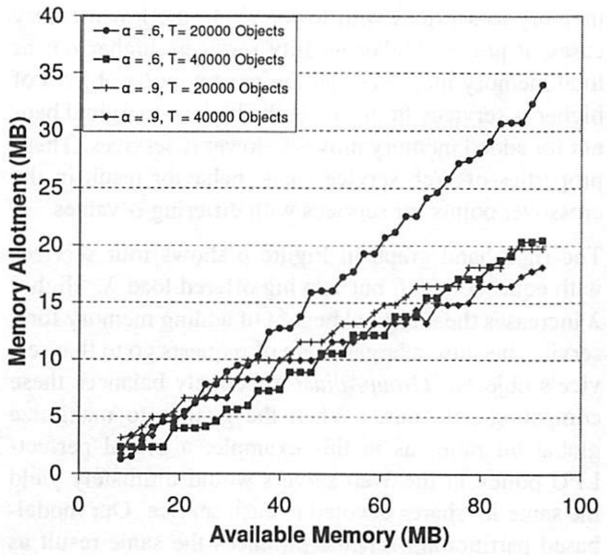


Figure 5: Using *Candidate* and *LocalAdjust* to determine memory and storage allotments for a service. As service load (λ) increases, *Candidate* increases the storage share ϕ to meet response time targets. After encountering a resource constraint at $\phi = 200$ IOPS, *LocalAdjust* transforms the candidate allotments to stay on target by adding memory instead.

Figure 5 also shows how the provisioning scheme adjusts the vectors to conform to a resource constraint. This example constrains ϕ to 200 IOPS, ultimately forcing the system to meet its targets by increasing the candidate M. *Candidate* itself does not consider resource constraints; the next two primitives adapt allotment vectors on a local or group basis to conform to resource constraints, or to allocate surplus resources to improve performance according to system-wide metrics.

4.2 Local Constraint or Surplus

The input to *LocalAdjust* is a candidate allotment vector and request arrival rate λ, together with specified constraints on each resource. The output of *LocalAdjust* is an adjusted vector that achieves a predicted average-case response time as close as possible to the target, while conforming to the resource constraints. Since this paper focuses primarily on memory and storage resources, we ignore CPU constraints. Specifically, we assume that the expected CPU response time R_P for a given ρ_{target}



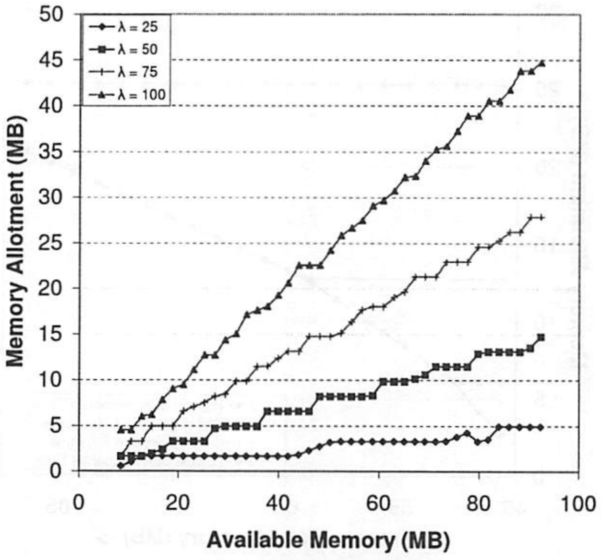


Figure 6: Memory allotments by *GroupAdjust* for four competing services with different caching profiles (left) and different request arrival rates (right).

is fixed and achievable. CPU allotments are relatively straightforward because memory and storage allotments affect per-request CPU demand only minimally. For example, if the CPU is the bottleneck, then the allotments for other resources are easy to determine: set λ to the saturation request throughput rate, and provision other resources as before.

If the storage constraint falls below the candidate storage allotment ϕ, then *LocalAdjust* assigns the maximum value to ϕ, and rebalances the system by expanding the memory allotment M to meet the response time target given the lower allowable request rate λ_S for storage. Determine the allowable $\lambda_S = \phi\rho_{target}$ at the preconfigured ρ_{target}. Determine the hit ratio H needed to achieve this λ_S using Equation (2), and the memory allotment M to achieve H using Equation (1).

Figure 5 illustrates the effect of *LocalAdjust* on the candidate vector under a storage constraint at $\phi = 200$ IOPS. As load λ increases, *LocalAdjust* meets the response time target by holding ϕ to the maximum and growing M instead. The candidate M varies in a (slightly) nonlinear fashion because H grows as M increases, so larger shares of the increases to λ are absorbed by the cache. This effect is partially offset by the dynamics of Web content caching captured in Equation (1): due to the nature of Zipf distributions, H grows logarithmically with M, requiring larger marginal increases to M to effect the same improvement in H.

If memory is constrained, *LocalAdjust* sets M to the maximum and rebalances the system by expanding ϕ (if possible). The algorithm is as follows: determine H and λ_S at M using Equations (1) and (2), and use λ_S to determine the adjusted storage allotment as $\phi = \lambda_S/\rho_{target}$. Then compensate for the reduced H by increasing ϕ further to reduce storage utilization levels below ρ_{target}, improving the storage response time R_S.

If both ϕ and M are constrained, assign both to their maximum values and report the predicted response time using the models in the obvious fashion. *LocalAdjust* adjusts allotments to consume a local surplus in the same way. This may be useful if the service is the only load component assigned to some server or set of servers. Surplus assignment is more interesting when the system must distribute resources among multiple competing services, as described below.

4.3 Group Constraint or Surplus

GroupAdjust adjusts a set of candidate allotment vectors to consume a specified amount of memory and/or storage resource. *GroupAdjust* may adapt the vectors to conform to resource constraints or to allocate a resource surplus to meet system-wide goals.

For example, *GroupAdjust* can reprovision available memory to maximize hit ratio across a group of hosted services. This is an effective way to allocate surplus memory to optimize global response time, to degrade service fairly when faced with a memory constraint, or to reduce storage loads in order to adapt to a storage constraint. Note that invoking *GroupAdjust* to adapt to constraints on specific shared storage units is an instance of storage-aware caching [16], achieved naturally as a side effect of model-based resource provisioning.

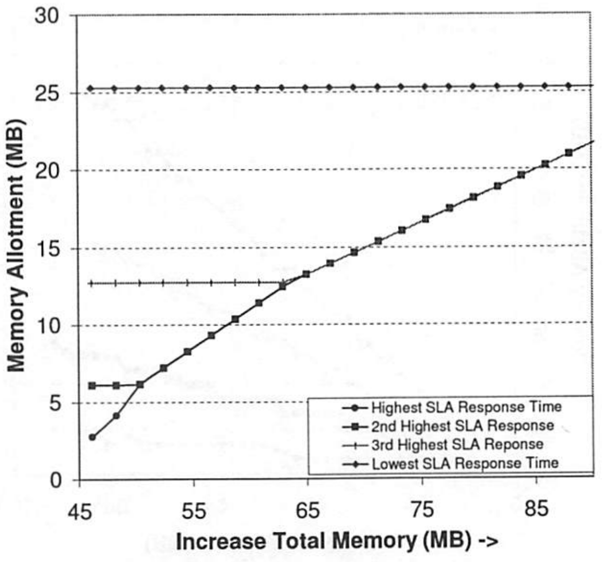


Figure 7: Using *Candidate* and *GroupAdjust* to allocate memory to competing services; in this example, the services are identical except for differing SLA response time targets. *Candidate* determines the minimum memory to meet each service's target; *GroupAdjust* allocates any surplus to the services with the least memory.

GroupAdjust assigns available memory to maximize hit ratio across a group as follows. First, multiply Equation (1) for each service by its expected request arrival rate λ. This gives a weighted value of service hit rates (hits or bytes per unit of time) for each service, as a function of its memory allotment M. The derivative of this function gives the marginal benefit for allocating a unit of memory to each service at its current allotment M. A closed form solution is readily available, since Equation (1) was obtained by integrating over the Zipf probability distribution (see Section 3.1). *GroupAdjust* uses a gradient-climbing approach (based on the Muse MSRP algorithm [12]) to assign each unit of memory to the service with the highest marginal benefit at its current M. This algorithm is efficient: it runs in time proportional to the product of the number of candidate vectors to adjust and the number of memory units to allocate.

To illustrate, Figure 6 shows the memory allotments of *GroupAdjust* to four competing services with different cache behavior profiles (α, S, T), as available memory increases on the x-axis. The left-hand graph varies the α and T parameters, holding offered load λ constant across all services. Services with higher α concentrate more of their references on the most popular objects, improving cache effectiveness for small M, while services with lower T can cache a larger share of their objects with a given M. The graph shows that for services with the same α values, *GroupAdjust* prefers to allocate memory to services with lower T. In the low-memory cases, it prefers higher-locality services (higher α); as total memory increases and the most popular objects of higher α services fit in cache, the highest marginal benefit for added memory moves to lower α services. These properties of Web service cache behavior result in the crossover points for services with differing α values.

The right-hand graph in Figure 6 shows four services with equal α and T but varying offered load λ. Higher λ increases the marginal benefit of adding memory for a service, because a larger share of requests go to that service's objects. *GroupAdjust* effectively balances these competing demands. When the goal is to maximize global hit ratio, as in this example, a global perfect-LFU policy in the Web servers would ultimately yield the same M shares devoted to each service. Our model-based partitioning scheme produces the same result as the reactive policy (if the models are accurate), but it also accommodates other system goals in a unified way. For example, *GroupAdjust* can use the models to optimize for global response time in a storage-aware fashion; it determines the marginal benefit in response time for each service as a function of its M, factoring in the reduced load on shared storage. Similarly, it can account for service priority by optimizing for "utility" [12, 21] or "yield" [32], composing the response time function for each service with a yield function specifying the value for each level of service quality.

4.4 Putting It All Together

Candidate and *GroupAdjust* work together to combine differentiated service with other system-wide performance goals. Figure 7 illustrates how they allocate surplus memory to optimize global response time for four example services. In this example, the hosted services have identical λ and caching profiles (α, S, T), but their SLAs specify different response time targets. *Candidate* allocates the minimum memory (total 46 MB) to meet the targets, assigning more memory to services with more stringent targets. *GroupAdjust* then allocates surplus memory to the services that offer the best marginal improvement in overall response time. Since the services have equivalent behavior, the surplus goes to the services with the smallest allotments.

Figure 8 further illustrates the flexibility of MBRP to adapt to observed changes in load or system behavior. It shows allotments $[M, \phi]$ for three competing services **s0**, **s1**, and **s2**. The system is configured to consolidate all loads on a shared server and storage unit, meet minimal response time targets when possible, and use any surplus resources to optimize global response time. The services begin with identical arrival rates λ, caching profiles (α, S, T), and response time targets. The experi-

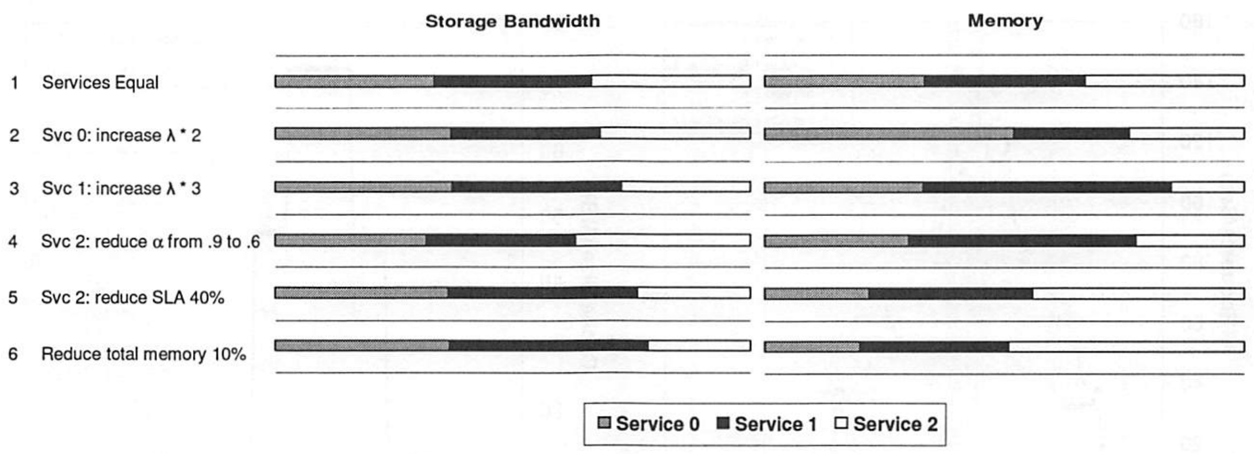


Figure 8: Allotments $[M, \phi]$ for three competing services in a multiple step experiment. For each step, the graphs represent the allotment percentage of each resource received by each service.

ment modifies a single parameter in each of six steps and shows its effect on the allotments; the changes through the steps are cumulative.

- In the base case, all services receive equal shares.
- Step 2 doubles λ for **s0**. The system shifts memory to **s0** to reflect its higher share of the request load, and increases its ϕ to rebalance storage utilization to ρ_{target}, compensating for a higher storage load.
- Step 3 triples the arrival rate for **s1**. The system shifts memory to **s1**; note that the memory shares match each service's share of the total request load. The system also rebalances storage by growing **s1**'s share at the expense of the lightly loaded **s2**.
- Step 4 reduces the cache locality of **s2** by reducing its α from .9 to .6. This forces the system to shift resources to **s2** to meet its response time target, at the cost of increasing global average response time.
- Step 5 lowers the response time target for **s2** by 40%. The system further compromises its global response time goal by shifting even more memory to **s2** to meet its more stringent target. The additional memory reduces the storage load for **s2**, allowing the system to shift some storage resource to the more heavily loaded services.
- The final step in this experiment reduces the amount of system memory by 10%. Since **s2** holds the minimum memory needed to meet its target, the system steals memory from **s0** and **s1**, and rebalances storage by increasing **s1**'s share slightly.

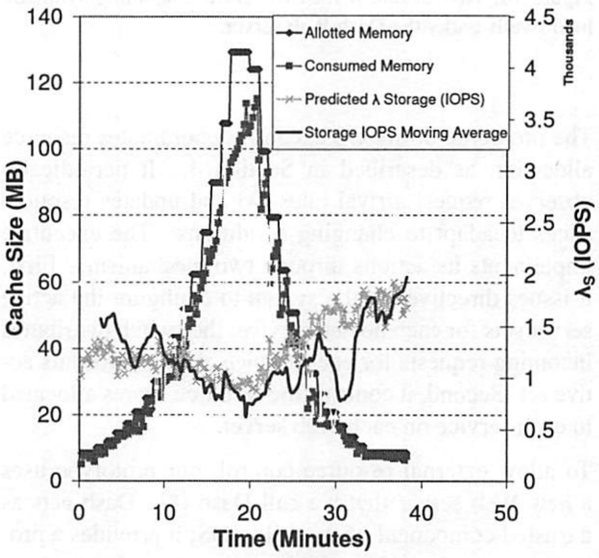


Figure 9: Predicted and observed storage I/O request rate vary with changing memory allotment for a Dash server under a typical static Web load (a segment of a 2001 trace of *www.ibm.com*).

5 Prototype and Results

To evaluate the MBRP approach, we prototyped key components of a Web service utility (as depicted in Figure 1) and conducted initial experiments using Web traces and synthetic loads. The cluster testbed consists of load generating clients, a reconfigurable L4 redirecting switch (from [12]), Web servers, and network storage servers accessed using the Direct Access File System protocol (DAFS [13, 24]), an emerging standard for network storage in the data center. We use the DAFS implementation from [24] over an Emulex cLAN network.

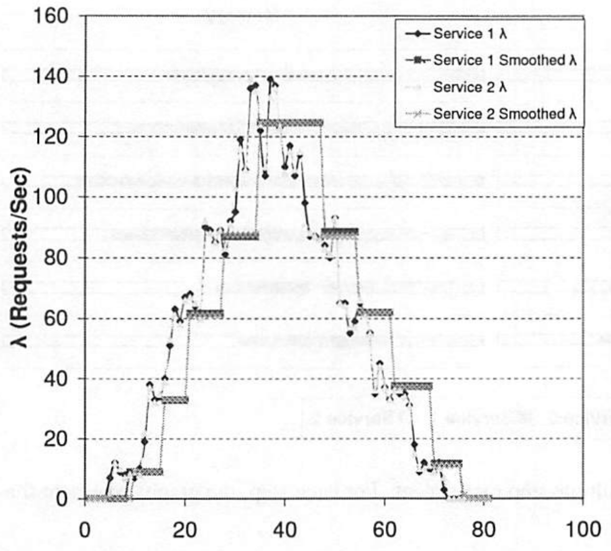


Figure 10: Arrival rate λ for two services handling synthetic load swells under the Dash Web server.

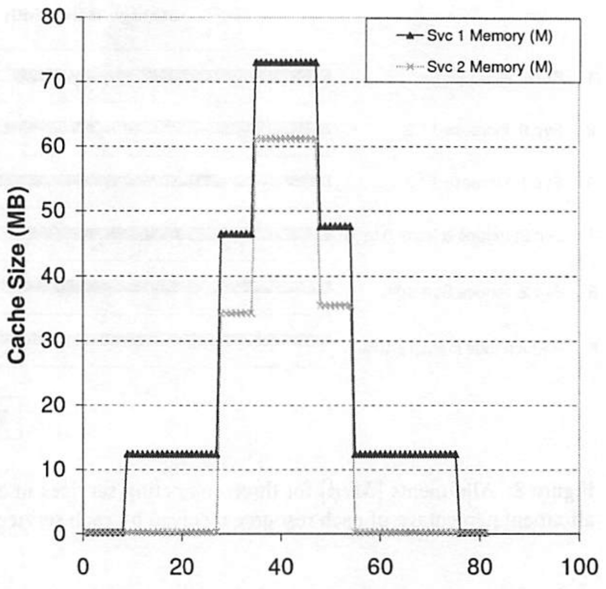


Figure 11: Memory allotments M for two services handling synthetic load swells under the Dash Web server. As load increases, *LocalAdjust* provisions additional memory to the services to keep the response time within SLA limits. Service 1 is characterized by higher storage access costs and therefore receives more memory to compensate.

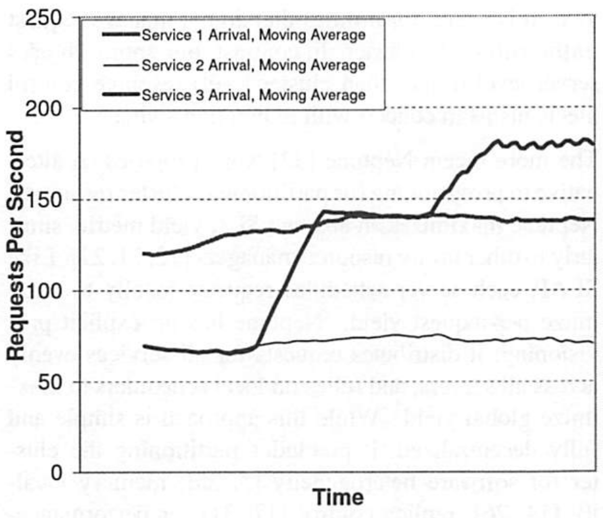
The prototype utility OS executive coordinates resource allocation as described in Section 4. It periodically observes request arrival rates (λ) and updates resource slices to adapt to changing conditions. The executive implements its actions through two mechanisms. First, it issues directives to the switch to configure the active server sets for each hosted service; the switch distributes incoming requests for each service evenly across its active set. Second, it controls the resource shares allocated to each service on each Web server.

To allow external resource control, our prototype uses a new Web server that we call Dash [8]. Dash acts as a trusted component of the utility OS; it provides a protected, resource-managed execution context for services, and exports powerful resource control and monitoring interfaces to the executive. Dash incorporates a DAFS user-level file system client, which enables user-level resource management in the spirit of Exokernel [19], including full control over file caching and and data movement [24]. DAFS supports fully asynchronous access to network storage, enabling a single-threaded, event-driven Web server structure as proposed in the Flash Web server work [27]—hence the name Dash. In addition, Dash implements a decentralized admission control scheme called Request Windows [18] that approximates proportional sharing of storage server throughput. The details and full evaluation of Dash and Request Windows are outside the scope of this paper.

For our experiments, the Dash and DAFS servers run on SuperMicro SuperServer 6010Hs with 866 MHz Pentium-III Xeon CPUs; the DAFS servers use one 9.1 GB 10,000 RPM Seagate Cheetah drive. Dash controls memory usage as reported in the experiments. Web traffic originates from a synthetic load generator ([10]) or Web trace replay as reported; the caching profiles (α, S, T) are known a priori and used to parameterize the models. All machines run FreeBSD 4.4.

We first present a simple experiment to illustrate the Dash resource control and to validate the hit ratio model (Equation (2)). Figure 9 shows the predicted and observed storage request rate λ_S in IOPS as the service's memory allotment M varies. The Web load is an accelerated 40-minute segment of a 2001 IBM trace [12] with steadily increasing request rate λ. Larger M improves the hit ratio for the Dash server cache; this tends to reduce λ_S, although λ_S reflects changes in λ as well as hit ratio. The predicted λ_S approximates the observed I/O load; the dip at $t = 30$ minutes is due to a transient increase in request locality, causing an unpredicted transient improvement in cache hit ratio. Although the models tend to be conservative in this example, the experiment demonstrates the need for a safety margin to protect against transient deviations from predicted behavior.

To illustrate the system's dynamic behavior in storage-aware provisioning, we conducted an experiment with two services with identical caching profiles (α, S, T) and response time targets, serving identical synthetic load swells on a Dash server. The peak IOPS through-



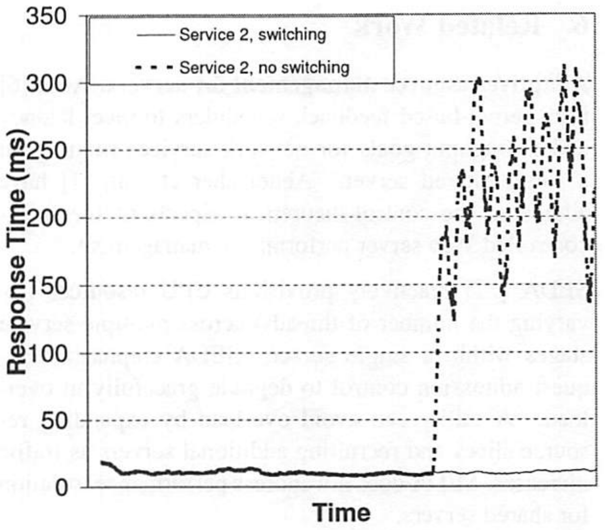


Figure 13: Arrival rates for competing services (left) and client response time with and without a bin-packing assignment phase to switch Web servers (right) handling the service.

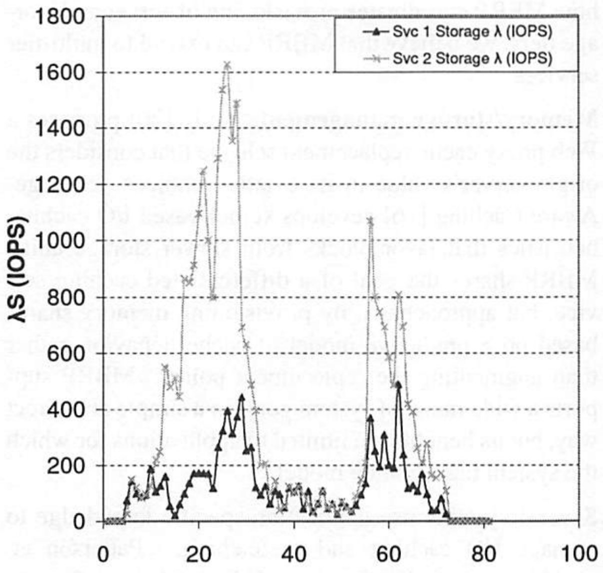


Figure 12: I/O rate (λ_S) for two services handling synthetic load swells under the Dash Web server. As additional memory is allocated to the services to meet SLA targets, the storage arrival rate decreases. Service 1 storage load reduces at a greater rate due to the additional memory allocated to this service.

puts available at the storage server for each service (reflected in the μ_s parameters) are constrained at different levels, with a more severe constraint for service 1. Figure 10 shows the arrival rates λ and the values smoothed by a "flop-flip" stepped filter [12] for input to the executive. Figure 11 shows the memory allotments for each service during the experiments, and Figure 12 shows the resulting storage loads λ_S. The storage constraints force the system to assign each service more memory to meet its target; as load increases, it allocates proportionally more memory to service 1 because it requires a higher H to meet the same target. As a result, service 1 shows a lower I/O load on its more constrained storage server. This is an example of how the model-based provisioning policies (here embodied in *LocalAdjust*) achieve similar goals to storage-aware caching [16].

The last experiment uses a rudimentary assignment planner to illustrate the role of assignment in partitioning cluster resources for response time targets. We compared two runs of three services on two Dash servers under the synthetic loads shown on the left-hand side of Figure 13, which shows a saturating load spike for service 3. In the first run, service 1 is bound to server A and services 2 and 3 are bound to server B. This results in a response time jump for service 2, shown in the right-hand graph in Figure 13; since the system cannot meet targets for both services, it uses *GroupAdjust* to provision B's resources for the best average-case response time. The second run employs a simple bin-packing scheme to assign the provisioned resource slices to servers. In this run, the system reassigns service 2 to A when the load spike for service 3 exposes the local resource constraint on B; this is possible because *Candidate* determines that there are sufficient resources on A to meet the response time targets for both services 1 and 2. To implement this choice, the executive directs the switch to route requests for service 2 to A rather than B. This allows service 2 to continue meeting its target. This simple example shows the power of the model-based provisioning primitives as a foundation for comprehensive resource management for cluster utilities.

6 Related Work

Adaptive resource management for servers. Aron [6] uses kernel-based feedback schedulers to meet latency and throughput goals for network services running on a single shared server. Abdelzaher et. al. [1] have addressed the control-theoretical aspects of feedback-controlled Web server performance management.

SEDA [37] reactively provisions CPU resources (by varying the number of threads) across multiple service stages within a single server. SEDA emphasizes request admission control to degrade gracefully in overload. A utility can avoid overload by expanding resource slices and recruiting additional servers as traffic increases. SEDA does not address performance isolation for shared servers.

Resource management for cluster utilities. Several studies use workload profiling to estimate the resource savings of multiplexing workloads in a shared utility. They focus on the probability of exceeding performance requirements for various degrees of CPU overbooking [4, 6, 34]. Our approach varies the degree of overbooking to adapt to load changes, but our current models consider only average-case service quality within each interval. The target utilization parameters (ρ_{target}) allow an external policy to control the "headroom" to handle predicted load bursts with low probability of violating SLA targets.

There is a growing body of work on adaptive resource management for cluster utilities under time-varying load. Our work builds on Muse [12], which uses a feedback-controlled policy manager and redirecting switch to partition cluster resources; [39] describes a similar system that includes a priority-based admission control scheme to limit load. Levy [21] presents an enhanced framework for dynamic SOAP Web services, based on a flexible combining function to optimize configurable class-specific and cluster-specific objectives. Liu [22] proposes provisioning policies (SLAP) for e-commerce traffic in a Web utility, and focuses on maximizing SLA profits. These systems incorporate analytical models of CPU behavior; MBRP extends them to provision for multiple resources including cluster memory and storage throughput.

Several of these policy-based systems rely on resource control mechanisms—such as Resource Containers [9] or VMware ESX [36]— to allow performance-isolated server consolidation. Cluster Reserves [7] extends a server resource control mechanism (Resource Containers) to a cluster. These mechanisms are designed to enforce a provisioning policy, but they do not define the policy. Cluster Reserves adjusts shares on individual servers to bound aggregate usage for each hosted service. It is useful for utilities that do not manage request traffic within the cluster. In contrast, our approach uses server-level (rather than cluster-level) resource control mechanisms in concert with redirecting switches.

The more recent Neptune [32] work proposes an alternative to provisioning (or partitioning) cluster resources. Neptune maximizes an abstract SLA yield metric, similarly to other utility resource managers [12, 21, 22]. Like SLAP, each server schedules requests locally to maximize per-request yield. Neptune has no explicit provisioning; it distributes requests for all services evenly across all servers, and relies on local schedulers to maximize global yield. While this approach is simple and fully decentralized, it precludes partitioning the cluster for software heterogeneity [5, 25], memory locality [14, 26], replica control [17, 31], or performance-aware storage placement [3].

Virtual Services [29] proposes managing services with multiple tiers on different servers. This paper shows how MBRP coordinates provisioning of server and storage tiers; we believe that MBRP can extend to multi-tier services.

Memory/storage management. Kelly [20] proposes a Web proxy cache replacement scheme that considers the origin server's value in its eviction choices. Storage-Aware Caching [16] develops kernel-based I/O caching heuristics that favor blocks from slower storage units. MBRP shares the goal of a differentiated caching service, but approaches it by provisioning memory shares based on a predictive model of cache behavior, rather than augmenting the replacement policy. MBRP supports a wide range of system goals in a simple and direct way, but its benefits are limited to applications for which the system has accurate models.

Several systems use application-specific knowledge to manage I/O caching and prefetching. Patterson et. al. [28] uses application knowledge and a cost/benefit algorithm to manage resources in a shared I/O cache and storage system. Faloutsos [15] uses knowledge of database access patterns to predict the marginal benefit of memory to reduce I/O demands for database queries.

Hippodrome [3] automatically assigns workload components to storage units in a utility data center. Our work is complementary and uses a closely related approach. Hippodrome is model-based in that it incorporates detailed models of storage system performance, but it has no model of the applications themselves; in particular, it cannot predict the effect of its choices on application service quality. Hippodrome employs a backtracking assignment planner (Ergastulum); we believe that a similar approach could handle server assignment in a utility data center using our MBRP primitives.

7 Conclusion

Scalable automation of large-scale network services is a key challenge for computing systems research in the next decade. This paper deals with a software framework and policies to manage network services as utilities whose resources are automatically provisioned and sold according to demand, much as electricity is today. This can improve robustness and efficiency of large services by multiplexing shared resources rather than statically overprovisioning each service for its worst case.

The primary contribution of this work is to demonstrate the potential of *model-based resource provisioning* (MBRP) for resource management in a Web hosting utility. Our approach derives from established models of Web service behavior and storage service performance. We show how MBRP policies can adapt to resource availability, request traffic, and service quality targets. These policies leverage the models to predict the performance effects of candidate resource allotments, creating a basis for informed, policy-driven resource allocation. MBRP is a powerful way to deal with complex resource management challenges, but it is only as good as the models. The model-based approach is appropriate for utilities running a small set of large-scale server applications whose resource demands are a function of load and stable, observable characteristics.

Acknowledgments

Darrell Anderson and Sara Sprenkle contributed to the ideas in this paper, and assisted with some experiments. We thank Mike Spreitzer, Laura Grit, and the anonymous reviewers for their comments, which helped to improve the paper.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, June 2001.

[2] D. C. Anderson and J. S. Chase. Fstress: A flexible network file service benchmark. Technical Report CS-2002-01, Duke University, Department of Computer Science, January 2002.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the First Usenix Conference on File and Storage Technologies (FAST)*, January 2002.

[4] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. Technical Report HPL-2002-339, HP Labs, November 2002.

[5] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[6] M. Aron. *Differentiated and Predictable Quality of Service in Web Server Systems.* PhD thesis, Department of Computer Science, Rice University, October 2000.

[7] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS 2000)*, pages 90–101, June 2000.

[8] O. M. Asad. Dash: A direct-access Web server with dynamic resource provisioning. Master's thesis, Duke University, Department of Computer Science, December 2002.

[9] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[10] P. Barford and M. E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of Performance '98/ACM SIGMETRICS '98*, pages 151–160, June 1998.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom '99*, March 1999.

[12] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[13] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[14] R. P. Doyle, J. S. Chase, S. Gadde, and A. M. Vahdat. The trickle-down effect: Web caching and server request distribution. *Computer Communications: Selected Papers from the Sixth International Workshop on Web Caching and Content Delivery (WCW)*, 25(4):345–356, March 2002.

[15] C. Faloutsos, R. T. Ng, and T. K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560, April 1995.

[16] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of the First Usenix Conference on File and Storage Technologies (FAST)*, January 2002.

[17] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[18] W. Jin, J. S. Chase, and J. Kaur. Proportional sharing for a storage utility. Technical Report CS-2003-02, Duke University, Department of Computer Science, January 2003.

[19] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[20] T. P. Kelly, S. Jamin, and J. K. MacKie-Mason. Variable QoS from shared Web caches: User-centered design and value-sensitive replacement. In *Proceedings of the MIT Workshop on Internet Service Quality Economics (ISQE 99)*, December 1999.

[21] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster-based Web services. In *Proceedings of the 8th International Symposium on Integrated Network Management (IM 2003)*, March 2003.

[22] Z. Liu, M. S. Squillante, and J. L. Wolf. On maximizing service-level-agreement profits. In *Proceedings of the ACM Conference on Electronic Commerce (EC'01)*, October 2001.

[23] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST)*, March 2003.

[24] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, R. Kisley, A. Gallatin, R. Wickremisinghe, and E. Gabber. Structure and performance of the Direct Access File System. In *USENIX Technical Conference*, pages 1–14, June 2002.

[25] J. Moore, D. Irwin, L. Grit, S. Sprenkle, and J. Chase. Managing mixed-use clusters with Cluster-on-Demand. Technical report, Duke University, Department of Computer Science, November 2002.

[26] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenopoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[27] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[28] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[29] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual Services: A new abstraction for server consolidation. In *Proceedings of the Usenix 2000 Technical Conference*, June 2000.

[30] J. Rolia, S. Singhal, and R. Friedrich. Adaptive Internet data centers. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR '00)*, July 2000.

[31] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: A highly scalable cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Kiawah Island, December 1999.

[32] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based Internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[33] P. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next-generation operating systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 1998.

[34] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[35] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. The potential costs and benefits of long-term prefetching for content distribution. *Computer Communications: Selected Papers from the Sixth International Workshop on Web Caching and Content Delivery (WCW)*, 25(4):367–375, March 2002.

[36] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[37] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

[38] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[39] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proceedings of IEEE Infocom 2001*, April 2001.

Conflict-Aware Scheduling for Dynamic Content Applications

Cristiana Amza[†], Alan L. Cox[†], Willy Zwaenepoel[‡]

[†]*Department of Computer Science, Rice University, Houston, TX, USA*
[‡]*School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland*
{amza, alc}@cs.rice.edu, willy.zwaenepoel@epfl.ch

Abstract

We present a new lazy replication technique, suitable for scaling the back-end database of a dynamic content site using a cluster of commodity computers. Our technique, called conflict-aware scheduling, *provides both throughput scaling and 1-copy serializability. It has generally been believed that this combination is hard to achieve through replication because of the growth of the number of conflicts. We take advantage of the presence in a database cluster of a* scheduler *through which all incoming requests pass. We require that transactions specify the tables that they access at the beginning of the transaction. Using that information, a conflict-aware scheduler relies on a sequence-numbering scheme to implement 1-copy serializability, and directs incoming queries in such a way that the number of conflicts is reduced.*

We evaluate conflict-aware scheduling using the TPC-W e-commerce benchmark. For small clusters of up to eight database replicas, our evaluation is performed through measurements of a web site implementing the TPC-W specification. We use simulation to extend our measurement results to larger clusters, faster database engines, and lower conflict rates.

Our results show that conflict-awareness brings considerable benefits compared to both eager and conflict-oblivious lazy replication for a large range of cluster sizes, database speeds, and conflict rates. Conflict-aware scheduling provides near-linear throughput scaling up to a large number of database replicas for the browsing and shopping workloads of TPC-W. For the write-heavy ordering workload, throughput scales, but only to a smaller number of replicas.

1 Introduction

This paper studies replication in database clusters [12, 17, 24] serving as back-ends in dynamic content sites.

Dynamic content sites commonly use a three-tier architecture, consisting of a front-end web server, an application server implementing the business logic of the site, and a back-end database (see Figure 1). The (dynamic) content of the site is stored in the database.

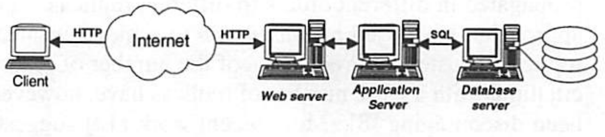


Figure 1: Common Architecture for Dynamic Content Sites

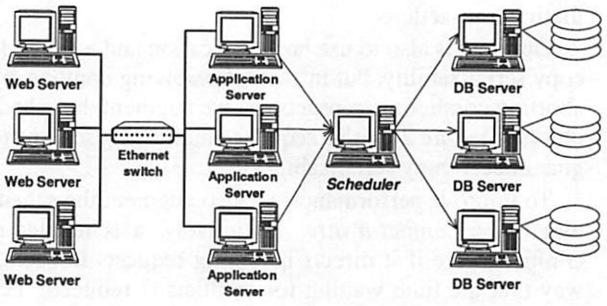


Figure 2: Clustering in a Dynamic Content Web Site

We focus in this paper on the case where the database back-end is the bottleneck. Practical examples of such situations may be found in e-commerce sites [1] or bulletin boards [19]. Bottlenecks in the web server tier are easily addressed by replicating the web server. Since it serves only static content, replication in this tier does not introduce any consistency issues. Addressing bottlenecks in the application server is a subject of further study.

Rather than following the common approach of using a very expensive high-end database machine as the back-end [6, 11], we investigate an alternative approach using a cluster of low-cost commodity machines. As Figure 2 shows, a *scheduler* is interposed between the application server(s) and the database cluster. The scheduler virtualizes the database cluster and makes it look to the application server as a single database. Therefore, no changes are required in the application server. Similarly, the methods presented in this paper do not require any changes to the database engine.

Based on observing the workload imposed on the database by dynamic content sites [1], and in particular by TPC-W [22], we argue that the nature of e-commerce workloads is such that lazy read-one, write-all replication [23] is the preferred way to distribute the data across the cluster. Furthermore, we argue that strong consistency (in technical terms, 1-copy serializability [4]) is desirable between the database replicas. From the combination of these two requirements – lazy replication and 1-copy serializability – stems the challenge addressed by this paper.

Lazy replication algorithms asynchronously propagate replica updates to other nodes, possibly even after the updating transaction commits. They do not provide 1-copy serializability, since, for instance, writes may be propagated in different orders to different replicas. Past approaches have used *reconciliation* to achieve eventual replica consistency. Predictions of the number of reconciliations with a large number of replicas have, however, been discouraging [8]. More recent work [12] suggests that for small clusters, both 1-copy-serializability and acceptable performance can be achieved by aborting conflicting transactions.

Our goal is also to use lazy replication and achieve 1-copy serializability, but instead of resolving conflicts by aborting conflicting transactions, we augment the scheduler (see Figure 2) with a sequence numbering scheme to guarantee 1-copy serializability.

To improve performance we also augment the scheduler to be *conflict-aware*. Intuitively, a scheduler is conflict-aware if it directs incoming requests in such a way that the time waiting for conflicts is reduced. For instance, a write on a data item may have been sent to all replicas, but it may have completed only at a subset of them. A conflict-aware scheduler keeps track of the completion status of each of the writes. Using this knowledge, it sends a conflicting read that needs to happen after this write to a replica where it knows the write has already completed.

A concern with a conflict-aware scheduler is the extra state and the extra computation in the scheduler, raising the possibility of it becoming the bottleneck or making it an issue in terms of availability and fault tolerance of the overall cluster. We present a lightweight implementation of a conflict-aware scheduler, that allows a single scheduler to support a large number of databases. We also demonstrate how to replicate the scheduler for availability and increased scalability.

Our implementation uses common software platforms such as the Apache Web server [3], the PHP scripting language [16], and the MySQL relational database [14]. As we are most interested in e-commerce sites, we use the TPC-W benchmark in our evaluation [22]. This benchmark specifies three workloads (browsing, shopping and ordering) with different percentages of writes in the workload. Our evaluation uses measurement of the implementation for small clusters (up to 8 database machines). We further use simulation to assess the performance effects of larger clusters, more powerful database machines, and varying conflict rates. Finally, in order to provide a better understanding of the performance improvements achieved by a conflict-aware scheduler, we have implemented a number of alternative schedulers that include only some of the features of the strategy implemented in conflict-aware scheduling.

Our results show that:

1. The browsing and shopping workloads scale very well, with close to linear increases in throughput up to 60 and 40 databases respectively. The ordering workload scales only to 16 databases.

2. The benefits of conflict awareness in the scheduler are substantial. Compared to a lazy protocol without this optimization, the improvements are up to a factor of 2 on our largest experimental platform, and up to a factor of 3 in the simulations. Compared to an eager protocol, the improvements are even higher (up to a factor of 3.5 in the experiments, and up to a factor of 4.4 in the simulations).

3. Simulations of more powerful database machines and varying conflict rates validate the performance advantages of conflict-aware scheduling on a range of software/hardware environments.

4. One scheduler is enough to support scaling for up to 60 MySQL engines for all workloads, assuming equivalent nodes are used for the scheduler and MySQL engine. Three schedulers are necessary if the databases become four times faster.

5. The cost of maintaining the extra state in the scheduler is minimal in terms of scaling, availability and fault tolerance.

6. Even with conflict avoidance, the eventual scaling limitations stem from conflicts.

The outline of the rest of the paper is as follows. Section 2 provides the necessary background on the characteristics of dynamic content applications. Section 3 introduces our solution. Section 4 describes our prototype implementation. Section 5 describes the fault tolerance aspects of our solution. Section 6 describes other scheduling techniques introduced for comparison with a conflict-aware scheduler. Section 7 presents our benchmark and experimental platform. We investigate scaling experimentally in Section 8, and by simulation in Section 9. Section 10 discusses related work. Section 11 concludes the paper.

2 Background and Motivation

In this section we provide an intuitive motivation for the methods used in this paper. This motivation is based on observing the characteristics of the workloads imposed on the database of a dynamic content site by a number of benchmarks [1], in particular by TPC-W [22].

First, there is a high degree of locality in the database access patterns. At a particular point in time, a relatively small working set captures a large fraction of the accesses. For instance, in an online bookstore, best-sellers are accessed very frequently. Similarly, the stories of the day on a bulletin board, or the active auctions on an auction site receive the most attention. With common application sizes, much of the working set can be captured in memory. As a result, disk I/O is limited. This characteristic favors replication as a method for distributing the data compared to alternative methods such as data partitioning. Replication is suitable to relieve hot spots, while data partitioning is more suitable for relieving high I/O demands [5].

Replication brings with it the need to define a consistency model for the replicated data. For e-commerce sites, 1-copy serializability appears appropriate [4]. With 1-copy serializability conflicting operations of different transactions appear in the same total order at all replicas. Enforcing 1-copy serializability avoids, for instance, a scenario produced by re-ordering of writes on different replicas in which on one replica it appears that one customer bought a particular item, while on a different replica it appears that another customer bought the same item.

Second, the computational cost of read queries is typically much larger than that of update queries. A typical update query consists of updating a single record selected by an equality test on a customer name or a product identifier. In contrast, read queries may involve complex search criteria involving many records, for instance "show me the ten most popular books on a particular topic that have been published after a certain date". This characteristic favors read-one, write-all replication: the expensive components of the workloads (reads) are executed on a single machine, and only the inexpensive components (writes) need to be executed on all machines. The desirability of read-one, write-all over more complex majority-based schemes has also been demonstrated under other workloads [12, 24].

Third, while locality of access is beneficial in terms of keeping data in memory, in a transactional setting it has the potential drawback of generating frequent conflicts, with the attendant cost of waiting for conflicts to be resolved. The long-running transactions present in the typical workloads may cause conflicts to persist for a long time. The frequency of conflicts also dramatically increases as a result of replication [8]. Frequent conflicts favor lazy replication methods, that asynchronously propagate updates [17, 21, 24]. Consider, for instance, a single-write transaction followed by a single-read transaction. If the two conflict, then in a conventional synchronous update protocol, the read needs to wait until the write has completed at all replicas. In contrast, in a lazy update protocol, the read can proceed as soon as the write has executed at its replica. If the two do not conflict, the read can execute in parallel with the write, and the benefits of asynchrony are much diminished.

Finally, frequent conflicts lead to increased potential for deadlock. This suggests the choice of a concurrency control method that avoids deadlock. In particular, we use conservative two-phase locking [4], in which all locks are acquired at the beginning of a transaction. Locks are held until the end of the transaction.

In summary, we argue that lazy read-one, write-all replication in combination with conservative two-phase locking is suitable for distributing data across a cluster of databases in a dynamic content site. In the next section, we show that by augmenting the scheduler with a sequence numbering scheme, lazy replication can be extended to provide 1-copy serializability. We also show how the scheduler can be extended to include conflict awareness, with superior performance as a result.

3 Design

3.1 Programming Model

A single (client) web interaction may include one or more transactions, and a single transaction may include one or more read or write queries. The application writer specifies where in the application code transactions begin and end. In the absence of transaction delimiters, each single query is considered a transaction and is automatically committed (so called "auto-commit" mode).

At the beginning of each transaction consisting of more than one query, the application writer inserts a lock acquire specifying all tables accessed in the transaction and their access types (read or write). This step is currently done by hand, but could be automated. Locks for single-operation transactions do not need to be specified.

3.2 Cluster Design

We consider a cluster architecture for a dynamic content site, in which a scheduler distributes incoming requests to a cluster of database replicas and delivers the responses to the application servers (see Figure 2). The scheduler may itself be replicated for performance or for availability. The presence of the scheduler is transparent to the application server and the database, both of which are unmodified.

If there is more than one scheduler in a particular configuration, the application server is assigned a particular scheduler at the beginning of a client web interaction. This assignment is currently done by round-robin. For each operation in a particular client web interaction, the application server only interacts with this single scheduler, unless the scheduler fails. These interactions are synchronous: for each query, the execution of the business logic for this particular client web interaction in the application server waits until it receives a response from the scheduler.

The application server sends the scheduler lock requests for multiple-operation transactions, reads, writes, commits and aborts.

3.3 Lazy Read-one, Write-all Replication

When the scheduler receives a lock request, a write or a commit from the application server, it sends it to all replicas and returns the response as soon as it receives a response from any of the replicas. Reads are sent only to a single replica, and the response is sent back to the the application server as soon as it is received from that replica.

3.4 1-Copy Serializability

The scheduler maintains 1-copy serializability by assigning a unique sequence number to each transaction. This assignment is done at the beginning of the transaction. For a multiple-operation transaction the sequence number is assigned when that transaction's lock request arrives at the scheduler. For a single-operation transaction, sequence number assignment is done when the transaction arrives at the scheduler. Lock requests are sent to all replicas, executed in order of their assigned sequence numbers, and held until commit, thus forcing all conflicting operations to execute in a total order identical at all replicas, and thus enforcing 1-copy serializability.

Transactions consisting of a single read query are treated differently. A single-read transaction holds locks only on the single replica where it executes. This optimization results in a very substantial performance improvement without violating 1-copy serializability.

3.5 Conflict-Aware Scheduling

Due to the asynchrony of replication, at any given point, some replicas may have fallen behind with the application of writes. Furthermore, some replicas may have not been able to acquire the locks for a particular transaction, due to conflicts. For reads, other than reads in single-read transactions, the scheduler first determines the set of replicas where the locks for its enclosing transaction have been acquired and where all previous writes in the transaction have completed. It then selects the least loaded replica from this set as the replica to receive the read query. The scheduler tries to find conflict-free replicas for single-read transactions as well, but may not be able to find one.

Conflict-aware scheduling requires that the scheduler maintains the completion status of lock requests and writes, for all database replicas.

4 Implementation

4.1 Overview

The implementation consists of three types of processes: scheduler processes (one per scheduler machine), a sequencer process (one for the entire cluster), and database proxy processes (one for each database replica). The sequencer assigns a unique sequence number to each transaction and thereby implicitly to each of its locks. A database proxy regulates access to its database server by letting an operation proceed only if the database has already processed all conflicting operations that precede it in sequence number order and all operations that precede it in the same transaction. The schedulers form the core of the implementation. They receive the various operations from the application servers, forward them to one or more of the database proxies, and relay the responses back to the application servers. The schedulers also interact with the sequencer to obtain a sequence number for each transaction.

In the following, we describe the state maintained at the scheduler and at the database proxy to support failure-free execution, and the protocol steps executed on receipt of each type of operation and its response. Additional state maintained for fault-tolerance purposes is described in Section 5.

4.2 The Scheduler's State

The scheduler maintains for each active transaction its sequence number and the locks requested by that transaction. In addition, it maintains a record for each operation that is outstanding with one or more database proxies. A record is created when an operation is received from the application server, and updated when it is sent to the database engines, or when a reply is received from one of them. The record for a read operation is deleted as soon as the response is received and delivered to the application server. For every replicated operation (i.e., lock request, write, commit or abort), the corresponding record is deleted only when all databases have returned a response.

The scheduler records the current load of each database (see section 4.10). This value is updated with new information included in each reply from a database proxy.

4.3 The Database Proxy's State

The database proxy maintains a reader-writer lock [23] queue for each table. These lock queues are maintained in order of sequence numbers. Furthermore, the database proxy maintains transaction queues, one per transaction, and an out-of-order queue for all operations that arrive out of sequence number order.

For each transaction queue, a head-of-queue record maintains the current number of locks granted to that transaction. Each transaction queue record maintains the operation to be executed. Transaction queue records are maintained in order of arrival.

4.4 Lock Request

For each lock request, the scheduler obtains a sequence number from the sequencer and stores this information together with the locks requested for the length of the transaction. The scheduler then tags the lock request with its sequence number and sends it to all database proxies. Each database proxy executes the lock request locally and returns an answer to the scheduler when the lock request is granted. The lock request is not forwarded to the database engine.

A lock request that arrives at the database proxy in sequence number order is split into separate requests for each of the locks requested. When all locks for a particular transaction have been granted, the proxy responds to the scheduler. The scheduler updates its record for that transaction, and responds to the application server, if this is the first response to the lock request for that transaction. A lock request that arrives out-of-order (i.e., does not have a sequence number that is one more than the last processed lock request) is put in the out-of-order queue. Upon further lock request arrivals, the proxy checks whether any request from the out-of-order queue can now be processed. If so, that request is removed from the out-of-order queue and processed as described above.

4.5 Reads and Writes

As the application executes the transaction, it sends read and write operations to the scheduler. The scheduler tags each operation with the sequence number that was assigned to the transaction. It then sends write operations to all database proxies, while reads are sent to only one database proxy.

The scheduler sends each read query to one of the replicas where the lock request and the previous writes of the enclosing transaction have completed. If more than one such replica exists, the scheduler picks the replica with the lowest load.

The database proxy forwards a read or write operation to its database only when all previous operations in the same transaction (including lock requests) have been executed. If an operation is not ready to execute, it is queued in the corresponding transaction queue.

4.6 Completion of Reads and Writes

On the completion of a read or a write at the database, the database proxy receives the response and forwards it to the scheduler. The proxy then submits the next operation waiting in the transaction queue, if any.

The scheduler returns the response to the application server if this is the first response it received for a write query or if it is the response to a read query. Upon receiving a response for a write from a database proxy, the scheduler updates its corresponding record to reflect the reply.

4.7 Commit/Abort

The scheduler tags the commit/abort received from the application server with the sequence number and locks requested at the start of the corresponding transaction, and forwards the commit/abort to all replicas.

If other operations from this transaction are pending in the transaction queue, the commit/abort is inserted at the tail of the queue. Otherwise, it is submitted to the database. Upon completion of the operation at the database, the database proxy releases each lock held by the transaction, and checks whether any lock requests in the queues can be granted as a result. Finally, it forwards the response to the scheduler.

Upon receiving a response from a database proxy, the scheduler updates the corresponding record to reflect the reply. If this is the first reply, the scheduler forwards the response to the application server.

4.8 Single-Read Transactions

The read is forwarded to a database proxy, where it executes after previous conflicting transactions have finished. In particular, requests for individual locks are queued in the corresponding lock queues, as with any other transaction, and the transaction is executed when all of its locks are available.

To choose a replica for the read, the scheduler first selects the set of replicas where the earlier update transactions in the same client web interaction, if any, have completed. It then determines the subset of this set at which no conflicting locks are held. This set may be empty. It selects a replica with the lowest load in the latter set, if it is not empty, and otherwise from the former set.

4.9 Single-Update Transactions

Single-update transactions are logically equivalent to multiple-operation transactions, but in the implementation they need to be treated a little differently; the necessary locks are not specified by the application logic, hence there is no explicit lock request. When the scheduler receives a single-update transaction, it computes the necessary locks and obtains a sequence number for the transaction. The transaction is then forwarded to all replicas with that additional information. Thus, the lock request is implicit rather than sent in a separate lock request message to the database proxy, but otherwise the database proxy treats a single-update transaction in the same way as any multiple-update transaction.

4.10 Load Balancing

We use the *Shortest Execution Length First (SELF)* load balancing algorithm. We measure off-line the execution time of each query on an idle machine. At runtime, the scheduler estimates the load on a replica as the sum of the (apriori measured) execution times of all queries outstanding on that back-end. SELF tries to take into account the widely varying execution times for different query types. The scheduler updates the load estimate for each replica with feedback provided by the database proxy in each reply. We have shown elsewhere [2] that SELF outperforms round-robin and shortest-queue-first algorithms for dynamic content applications.

5 Fault Tolerance and Data Availability

5.1 Fault Model

For ease of implementation, we assume a fail-stop fault model. However, our fault tolerance algorithm could be generalized to more complex fault models.

5.2 Fault Tolerance of the Sequencer

At the beginning of each transaction, a scheduler requests a sequence number from the sequencer. Afterwards, the scheduler sends to all other schedulers a replicate_start_transaction message containing the sequence number for this transaction, and waits for an acknowledgment from all of them. All other schedulers create a record with the sequence number and the coordinating scheduler of this transaction. No disk logging is done at this point.

If the sequencer fails, replication of the sequence numbers on all schedulers allows for restarting the sequencer with the last sequence number on another machine. In case all schedulers fail, the sequence numbers are reset everywhere (sequencer, schedulers, database proxies).

5.3 Atomicity and Durability of Writes

To ensure that all writes are eventually executed, regardless of any sequence of failures in the schedulers and the databases, each scheduler keeps a persistent log of all write queries of committed transactions that it handled, tagged with the transaction's sequence number. The log is kept per table in sequence number order, to facilitate recovery (see below). To add data availability, the scheduler replicates this information in the memory of all other schedulers. Each database proxy maintains the sequence number for the last write it committed on each table. The database proxy does not log its state to disk.

In more detail, before a commit query is issued to any database, the scheduler sends a replicate_end_transaction message to the other schedulers. This message contains the write queries that have occurred during the transaction, and the transaction's sequence number. All other schedulers record the writes, augment the corresponding remote transaction record with the commit decision, and respond with an acknowledgment. The originator of the replicate_end_transaction message waits for the acknowledgments, then logs the write queries to disk. After the disk logging has completed, the commit is issued to the database replicas. For read-only transactions, the commit decision is replicated but not logged to disk.

5.3.1 Scheduler Failure

In the case of a single scheduler failure, all transactions of the failed scheduler for which the live schedulers do not have a commit decision are aborted. A transaction for which a commit record exists, but for which a database proxy has not yet received the commit decision is aborted at that particular replica, and then its writes are replayed. The latter case is, however, very rare.

In more detail, a fail-over scheduler contacts all available database proxies. The database proxy waits until all queued operations finish at its database, including any pending commits. The proxy returns to the scheduler the sequence number for the last committed write on each database table, and the highest sequence number of any lock request received by the database proxy. The fail-over scheduler determines all the failed scheduler's transactions for which a commit record exists and for which a replica has not committed the transaction. The reason that a replica has not committed a transaction may be either that it did not receive the transaction's lock request or that it did not receive the commit request. The first case is detected by the sequence number of the transaction being larger than the highest lock sequence number received by the proxy. In this case, all the transaction's writes and its commit are replayed to the proxy. In the second case, the fail-over scheduler first aborts the transaction and then sends the writes and the commit. The database proxy also advances its value of the highest lock sequence number received and its value of the last sequence number of a committed write on a particular table, as appropriate.

For all other transactions handled by the failed scheduler, the fail-over scheduler sends an abort to all database proxies. The database proxies rollback the specified transactions, and also fill the gap in lock sequence numbers in case the lock with the sequence number specified in the abort was never received. The purpose of this, is to let active transactions of live schedulers with higher sequence numbered locks proceed.

Each scheduler needs to keep logs for committed writes and records of assigned lock sequence numbers until all replicas commit the corresponding transactions. Afterwards, these records can be garbage-collected.

When a scheduler recovers, it contacts another scheduler and replicates its state. In the rare case where all schedulers fail, the state is reconstructed from the disk logs of all schedulers. All active transactions are aborted on all database replicas, the missing writes are applied on all databases, and all sequence numbers are reset everywhere.

5.3.2 Network Failure

To address temporary network connection failures, each database proxy can send a "selective retransmission" request for transactions it has missed. Specifically, the database proxy uses a timeout mechanism to detect gaps in lock sequence that have not been filled in a given period of time. It then contacts an available scheduler and provides its current state. The scheduler rolls forward the database proxy including updating its highest lock sequence number.

5.3.3 Database Failure

When a database recovers from failure, its database proxy contacts all available schedulers and selects one scheduler to coordinate its recovery. The coordinating scheduler instructs the recovering database to install a current database snapshot from another replica, with its current state. Each scheduler re-establishes its connections to the database proxy and adds the replica to its set of available machines. The scheduler starts sending to the newly incorporated replica at the beginning of the next transaction. Afterwards, the database proxy becomes up-to-date by means of the selective retransmission requests as described in the case of network failure.

In addition, each database proxy does periodic checkpoints of its database together with the current state (in terms of the last sequence numbers of its database tables). To make a checkpoint, the database proxy stops all write operations going out to the database engine, and when all pending write operations have finished, it takes a snapshot of the database and writes the new state. If any tables have not changed since the last checkpoint, they do not need to be included in the new checkpoint. Checkpointing is only necessary to support recovery in the unlikely case where all database replicas fail. In this case, each database proxy reinstalls the database from its own checkpoint containing the database snapshot of the last known clean state. Subsequently, recovery proceeds as in the single database failure case above.

6 Algorithms Used for Comparison

In this section, we introduce a number of other scheduling algorithms for comparison with conflict-aware scheduling. By gradually introducing some of the features of conflict-aware scheduling, we are able to demonstrate what aspects of conflict-aware scheduling contribute to its overall performance. Our first algorithm is an eager replication scheme that allows us to show the benefits of asynchrony. We then look at a number of scheduling algorithms for lazy replication. Our second algorithm is a conventional scheduler that chooses a fixed replica based on load at the beginning of the transaction and executes all operations on that replica. Our third algorithm is another fixed-replica approach, but it introduces one feature of conflict-aware scheduling: it chooses as the fixed replica the one that responds first to the transaction's lock request rather than the least loaded one. We then move away from fixed-replica algorithms, allowing different replicas to execute reads of the same transaction, as in conflict-aware scheduling. Our fourth and final scheduler chooses the replica with the lowest load at the time of the read, allowing us to assess the difference between this approach and conflict-aware scheduling, where a read is directed to a replica without conflicts.

We refer to these scheduler algorithms as Eager, FR-L (Fixed Replica based on Load), FR-C (Fixed Replica based on Conflict), and VR-L (Variable Replica based on Load). Using this terminology, the conflict-aware scheduler would be labeled VR-C (Variable Replica based on Conflict), but we continue to refer to it as the conflict-aware scheduler.

In all algorithms, we use the same concurrency control mechanism, i.e., conservative two-phase locking, the same sequence numbering method to maintain 1-copy serializability, and the same load balancing algorithm (see Section 4.10).

6.1 Eager Replication (Eager)

Eager follows the algorithm described by Weikum et al. [23], which uses synchronous execution of lock requests, writes and commits on all replicas. In other words, the scheduler waits for completion of every lock, write or commit operation on all replicas, before sending a response back to the application server. The scheduler directs a read to the replica with the lowest load.

6.2 Fixed Replica Based on Load (FR-L)

FR-L is a conventional scheduling algorithm, in which the scheduler is essentially a load balancer. At the beginning of the transaction, the scheduler selects the replica with the lowest load. It just passes through operations tagged with their appropriate sequence numbers. Lock requests, writes and commits are sent to all replicas, and the reply is sent back to the application server when the chosen replica replies to the scheduler. Reads are sent to the chosen replica. The FR-L scheduler needs to record only the chosen replica for the duration of each transaction.

6.3 Fixed Replica Based on Conflict (FR-C)

FR-C is identical to FR-L, except that the scheduler chooses the replica that first responds to the lock request as the fixed replica for this transaction. As in FR-L, all reads are sent to this replica, and a response is returned to the application server when this replica responds to a lock request, a write or a commit. The FR-C scheduler's state is also limited to the chosen replica for each transaction.

6.4 Variable Replica Based on Load (VR-L)

In the **VR-L** scheduler, the response to the application server on a lock request, write, or commit is sent as soon as the first response is received from any replica. A read is sent to the replica with the lowest load at the time the read arrives at the scheduler. This may result in the read being sent to a replica where it needs to wait for the completion of conflicting operations in other transactions or previous operations in its own transaction.

The VR-L scheduler needs to remember for the duration of each replicated query whether it has already forwarded the response and whether all machines have responded, but, unlike a conflict-aware scheduler, it need not remember which replicas have responded. In other words, the size of the state maintained is O(1) not O(N) in the number of replicas.

7 Experimental Platform

7.1 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council (TPC) [22] is a transactional web benchmark for e-commerce systems. The benchmark simulates a bookstore.

The database contains eight tables: customer, address, orders, order_line, credit_info, item, author, and country. The most frequently used are order_line, orders and credit_info, which contain information about the orders placed, and item and author, which contain information about the books. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100,000 items and 2.8 million customers which results in a database of about 4 GB. The inventory images, totaling 1.8 GB, reside on the web server.

We implemented the fourteen different interactions specified in the TPC-W benchmark specification. Six of the interactions are read-only, while eight cause the database to be updated. The read-only interactions include access to the home page, listing of new products and best-sellers, requests for product detail, and two interactions involving searches. Update transactions include user registration, updates of the shopping cart, two order-placement transactions, and two for administrative tasks. The frequency of execution of each interaction is specified by the TPC-W benchmark. The complexity of the interactions varies widely, with interactions taking between 20 and 700 milliseconds on an unloaded machine. The complexity of the queries varies widely as well. In particular, the most heavyweight read queries are 50 times more expensive than the average write query.

TPC-W uses three different workload mixes, differing in the ratio of read-only to read-write interactions. The browsing mix contains 95% read-only interactions, the shopping mix 80%, and the ordering mix 50%.

7.2 Client Emulation Software

We implemented a client-browser emulator that allows us to vary the load on the web site by varying the number of emulated clients. A client session is a sequence of interactions for the same client. For each client session, the client emulator opens a persistent HTTP connection to the web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another. The client session time and the think time are generated from a random distribution with a mean value specified in TPC-W.

7.3 Software Environment

We use three popular open-source software packages: the Apache web server [3], the PHP scripting language [16], and the MySQL database server [14]. Since PHP is implemented as an Apache module, the web server and application server co-exist on the same machine(s). We use Apache v.1.3.22 for the web server, configured with the PHP v.4.0.1 module. We use MySQL v.4.0.1 with InnoDB transactional extensions as our database server.

The schedulers and database proxies are both implemented with event-driven loops that multiplex requests and responses between the web server and the database replicas. We use FreeBSD's scalable kevent primitive [13] to efficiently handle thousands of connections at a single scheduler.

7.4 Hardware Environment

We use the same hardware for all machines, including those running the client emulation software, the web servers, the schedulers and the database engines. Each machine has an AMD Athlon 800Mhz processor running FreeBSD 4.1.1, 256MB SDRAM, and a 30GB ATA-66 disk drive. All machines are connected through a switched 100Mbps Ethernet LAN.

8 Experimental Results

The experimental results are obtained on a cluster with 1 to 8 database server machines. We use a number of web server machines large enough to make sure that the web server stage is not the bottleneck. The largest number of web server machines used for any experiment was 8. We use 2 schedulers to ensure data availability.

We measure throughput in terms of the number of web interactions per second (WIPS), the standard TPC-W performance metric. For a given number of machines we report the peak throughput. In other words, we vary the number of clients until we find the peak throughput and we report the average throughput number over several runs. We also report average response time at this peak throughput. Next, we break down the average query time into query execution time, and waiting

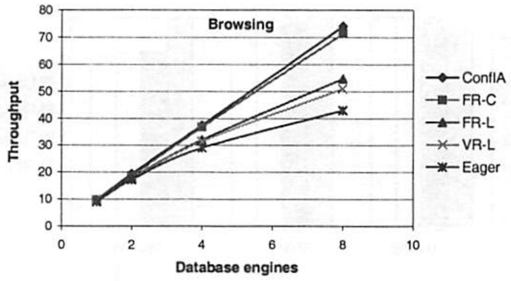


Figure 3: Throughput Comparison: The benefits of conflict avoidance and fine-grained scheduling for the browsing mix.

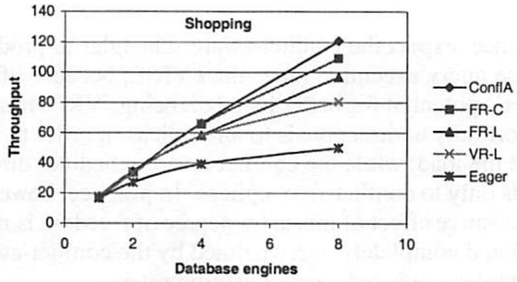


Figure 4: Throughput Comparison: The benefits of conflict avoidance and fine-grained scheduling for the shopping mix.

time (for locks and previous writes in the same transaction). Finally, we compare the performance in terms of throughput between the conflict-aware scheduler with and without fault tolerance and data availability.

8.1 Throughput

Figures 3 through 5 show the throughput of the various scheduling algorithms for each of the three workload mixes. In the x-axis we have the number of database machines, and in the y-axis the number of web interactions per second.

First, conflict-aware scheduling outperforms all other algorithms, and increasingly so for workload mixes with a large fraction of writes. Second, all asynchronous schemes outperform the eager scheme, again increasingly so as the fraction of writes increases. In particular,

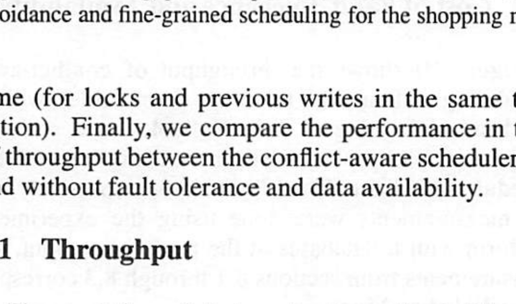


Figure 5: Throughput Comparison: The benefits of conflict avoidance and fine-grained scheduling for the ordering mix.

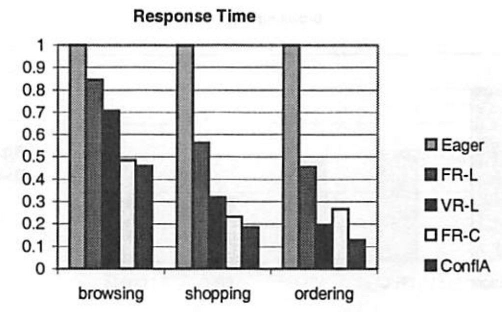


Figure 6: Web Interaction Response Time for All Schedulers and All Workloads, Normalized to Response Time of the Eager Scheduler

the conflict-aware protocol outperforms the eager protocol by factors of 1.7, 2.4 and 3.5 for browsing, shopping, and ordering, respectively at eight replicas. Third, for the fixed replica algorithms, choosing the replica by conflict (FR-C) rather than by load (FR-L) provides substantial benefits, a factor of 1.4, 1.4, and 1.25 for the largest configuration, for each of the three mixes, respectively. Fourth, variable replica algorithms provide better results than fixed replica algorithms, with the conflict-aware scheduler showing a gain of a factor of 1.5, 1.6 and 2, for browsing, shopping, and ordering, respectively, compared to FR-L, at 8 replicas. Finally, FR-C performs better than VR-L for the browsing and the shopping mix, but becomes worse for the ordering mix. Because of the larger numbers of reads in the browsing and shopping mixes, VR-L incurs a bigger penalty for these mixes by not sending the reads to a conflict-free replica, possibly causing them to have to wait. FR-C, in contrast, tries to send the reads to conflict-free replicas. In the ordering mix, reads are fewer. Therefore, this advantage for FR-C becomes smaller. VR-L's ability to shorten the write and commit operations by using the first response becomes the dominant factor.

8.2 Response Time

Figure 6 presents a comparison of the average web interaction response time for the five scheduler algorithms and the three workload mixes on an 8-database cluster at peak throughput. For each workload mix, the results are normalized to the average response time of the eager scheduler for that workload mix.

These results show that the conflict-aware scheduler provides better average response times than all other schedulers. The performance benefits of reducing conflict waiting time are reflected in response time reductions as well, with the same relative ranking for the different protocols as in the throughput comparison.

8.3 Breakdown of Query Time

Figures 7, 8, and 9, show a breakdown of the *query* response time into query execution time and waiting time.

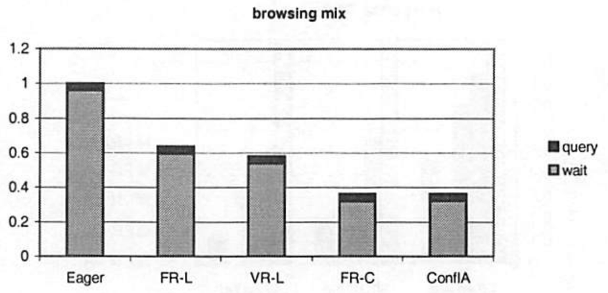


Figure 7: Query Time Breakdown for All Schedulers for the Browsing Mix, Normalized by the Query Time of the Eager Scheduler.

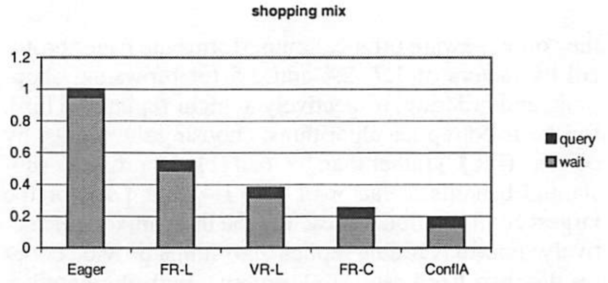


Figure 8: Query Time Breakdown for All Schedulers for the Shopping Mix, Normalized by the Query Time of the Eager Scheduler.

The waiting time is mostly due to conflicts; waiting for previous writes in the same transaction is negligible. For each workload mix, the results are normalized to the average query response time for the eager scheduler for that workload mix.

These results further stress the importance of reducing conflict waiting time. For all protocols, and all workloads, conflict waiting time forms the largest fraction of the query time. Therefore, the scheduler that reduces the conflict waiting time the most performs the best in terms of overall throughput and response time. The differences in query execution time between the different protocols are minor and do not significantly influence the overall throughput and response time. One might, for instance, expect the conflict-aware scheduler to produce worse query execution times than VR-L, because of the latter's potential for better load balancing. VR-L has the opportunity to direct reads to all replicas in order to balance the load, while the conflict-aware scheduler directs reads only to conflict-free replicas. In practice, however, the positive effect of this extra degree of freedom is minimal and completely overwhelmed by the conflict-aware scheduler's reduced conflict waiting times.

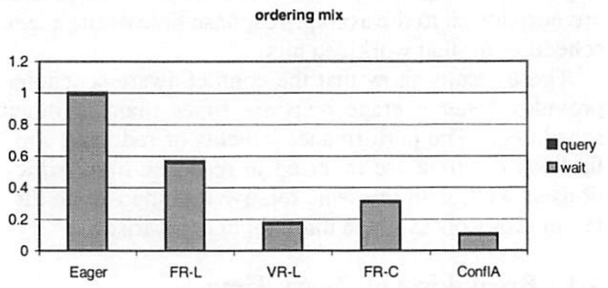


Figure 9: Query Time Breakdown for All Schedulers for the Ordering Mix, Normalized by the Query Time of the Eager Scheduler.

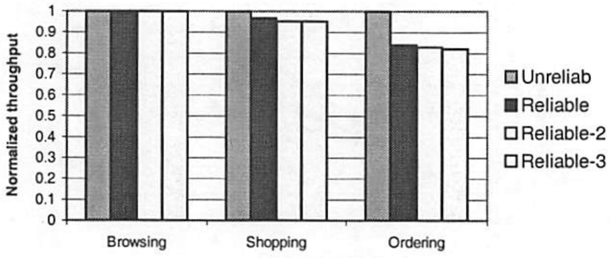


Figure 10: Overhead of Providing Fault tolerance and Various Degrees of Availability for the Conflict-Aware Scheduler for All Workload Mixes

8.4 Cost of Fault Tolerance and Availability

Figure 10 shows the throughput of conflict-aware scheduling without fault tolerance (Unreliable), with the overhead of logging to disk (Reliable), and with logging to disk plus replicating the state using two and three schedulers (Reliable-2 and Reliable-3, respectively). All the measurements were done using the experimental platform with 8 databases at the peak throughput. The measurements from sections 8.1 through 8.3 correspond to the Reliable-2 bar.

The overhead for fault tolerance and data availability is negligible for the browsing and shopping mixes and around 16% for the ordering mix. This overhead is mainly due to logging, with no extra overhead when replication of writes to remote schedulers is added to logging (as seen by comparing the Reliable and Reliable-2 bars).

A full checkpoint currently takes 5 minutes to perform for our 4 GB database. We did not include this overhead in our above measurements, because well known techniques for minimizing the time for taking file snapshots exist [10].

9 Simulations

We use simulation to extrapolate from our experimental results in three different directions. First, we explore how throughput scales if a larger cluster of database replicas is available. Second, we investigate the effect of faster databases, either by using faster database software or faster machines. Third, we show how the performance differences between the various schedulers evolve as the conflict rate is gradually reduced.

9.1 Simulator Design

We have developed two configurable simulators, one for the web/application server front-ends and the other for the database back-ends. We use these front-end and back-end simulators to drive actual execution of the schedulers and the database proxies. Our motivation for running the actual scheduler code and the actual database proxy code rather than simulating them is to measure their overheads experimentally and determine whether or not they may become a bottleneck for large clusters or faster databases.

The web server simulator models a very fast web server. It accepts client requests, and sends the corresponding queries to the scheduler without executing any application code. The scheduler and the database proxies perform their normal functions, but the database proxy sends each query to the database simulator rather than to the database.

The database simulator models a cluster of database replicas. It maintains a separate queue for each simulated replica. Whenever a query is received from the scheduler for a particular replica, a record is placed on that replica's queue. The simulator estimates a completion time for the query, using the same query execution time estimates as used for load balancing. It polls the queues of all replicas, and sends responses when the simulated time reaches the completion time for each query. The simulator does not model disk I/O. Based on profiling of actual runs, we estimate that the disk access time is mostly overlapped with computation, due to the locality in database accesses and the lazy database commits.

Calibration of the simulated system against measurement of the real 8-node database cluster shows that the simulated throughput numbers are within 12% of the experimental numbers for all three mixes.

9.2 Large Database Clusters

9.2.1 Results

We simulate all five schedulers for all three workload mixes for database cluster sizes up to 60 replicas. As with the experimental results, for a given number of replicas, we increase the number of clients until the system achieves peak throughput, and we report those peak throughput numbers. The results can be found in Figures 11, 12 and 13. In the x-axis we have the number of simulated database replicas, and in the y-axis the throughput in web interactions per second.

The simulation results show that the experimental results obtained on small clusters can be extrapolated to larger clusters. In particular, the conflict-aware scheduler outperforms all other schedulers, and the benefits of conflict awareness grow as the cluster size grows, especially for the shopping and the ordering mix. Furthermore, the relative order of the different schedulers remains the same, and, in particular, all lazy schemes out-

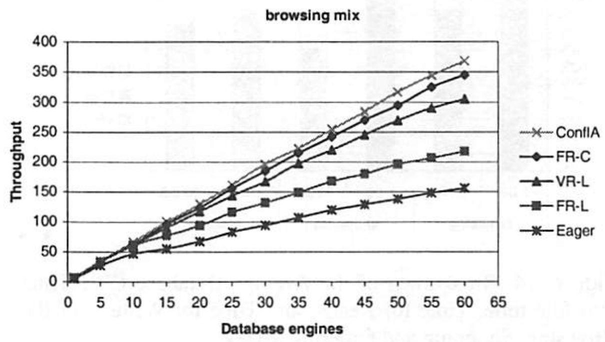


Figure 11: Simulated Throughput Results for the Browsing Mix

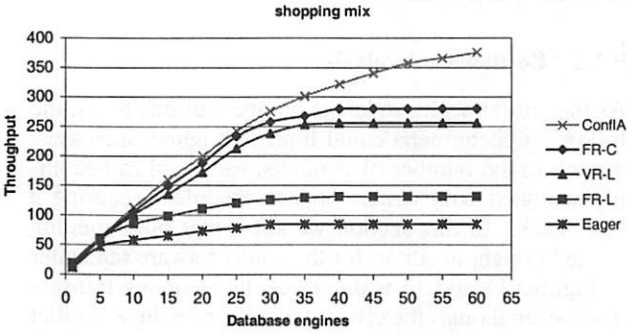


Figure 12: Simulated Throughput Results for the Shopping Mix

perform the eager scheduler (by up to a factor of 4.4 for the conflict-aware scheduler). The results for the shopping mix deserve particular attention, because they allow us to observe a flattening of the throughput of FR-C and VR-L as the number of machines grows, a phenomenon that we could not observe in the actual implementation. In contrast, throughput of the conflict-aware protocol continues to increase, albeit at a slower pace. With increasing cluster size, the number of conflicts increases [8]. Hence, choosing the replica based on a single criterion, either conflict (as in FR-C) or fine-grained

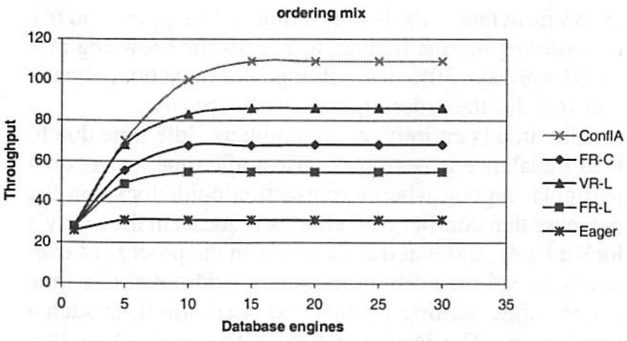


Figure 13: Simulated Throughput Results for the Ordering Mix

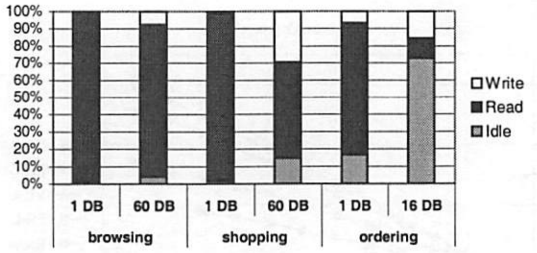


Figure 14: Breakdown of the Average Database CPU Time into Idle time, Time for Reads, and Time for Writes, for the Browsing, Shopping and Ordering Mixes.

load balancing (as in VR-L), is inferior to conflict-aware scheduling that combines both.

9.2.2 Bottleneck Analysis

As the cluster scales to larger numbers of machines, the following phenomena could limit throughput increases: growth in the number of conflicts, each replica becoming saturated with writes, or the scheduler becoming a bottleneck. In this section we show that the flattening of the throughput curves for the conflict-aware scheduler in Figures 12 and 13 is due to conflicts among transactions, even though the scheduler seeks to reduce conflict waiting time. A fortiori, for the other schedulers, which invest less effort in reducing conflicts, conflicts are even more of an impediment to good performance at large cluster sizes.

Using the conflict-aware scheduler, Figure 14 shows the breakdown of the average database CPU time into idle time, time processing reads, and time processing writes. The breakdown is provided for each workload, for one replica and for either the largest number of replicas simulated for that workload or for a number of replicas at which the throughput curve has flattened out.

For the browsing mix, which still scales at 60 replicas, idle time remains low even at that cluster size. For the shopping mix, which starts seeing some flattening out at 60 machines, idle time has grown to 15%. For the ordering mix, which does not see any improvement in throughput beyond 16 replicas, idle time has grown considerably, to 73%. The fraction of write (non-idle) time grows from under 1% for browsing and shopping and 6% for ordering on one replica, to 8% for the browsing mix (at 60 replicas), 30% in the shopping mix (at 60 replicas), and 16% for the ordering mix (at 16 replicas).

Idle time is entirely due to conflicts. Idle time due to load imbalance is negligible. Most idle time occurs on a particular replica when a transaction holds locks on the database that conflict with all lock requests in the proxy's lock queues, and that transaction is in the process of executing a read on a different replica. Additional idle time occurs while waiting for the next operation from such a transaction. The results in Figure 14 clearly show that idle time, and thus conflicts, is the primary impediment to scaling. Write saturation (a replica being fully occupied with writes) does not occur.

CPU (%)	Memory (MB)	Network (MB/sec)	Disk (MB/sec)
58%	6.3	3.8	0.021

Table 1: Resource Usage at the Scheduler for the Shopping Mix at 60 Replicas

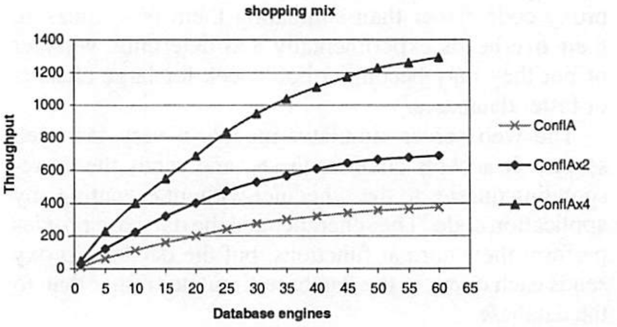


Figure 15: Simulated Throughput Results with Faster Database Replicas for the Shopping Mix

For the sizes of clusters simulated, the scheduler is not a bottleneck. While the scheduler CPU usage grows linearly with increases in cluster size, one scheduler is enough to sustain the maximum configuration for all three mixes. Table 1 shows the CPU, memory, disk and network usage at the scheduler for the shopping mix, in a configuration with one scheduler and 60 replicas. The CPU usage reaches about 58%, while all other resource usage is very low. For all other mixes, the resource usage is lower at their corresponding maximum configuration.

9.3 Faster Replicas

If the database is significantly faster, either by using more powerful hardware or by using a high-performance database engine, a conflict-aware scheduler continues to provide good throughput scaling. In Figure 15 we show throughput as a function of the number of replicas for databases twice and four times faster than the MySQL database we use in the experiments. [1] We simulate faster databases by reducing the estimated length of each query in the simulation. Figure 15 shows that the faster databases produce similar scaling curves with correspondingly higher throughputs. Three schedulers are necessary in the largest configuration with the fastest database.

9.4 Varying Conflict Rates

Table-level conservative two-phase locking, as used in our implementation, causes a high conflict rate. We investigate the benefits of conflict-aware scheduling under

[1] This is the highest speed of database for which we could simulate a cluster with 60 replicas.

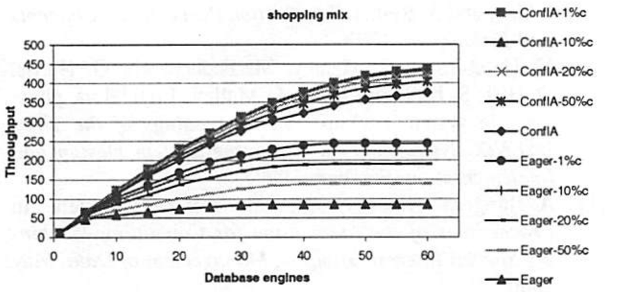


Figure 16: Simulated Throughput Results for Various Conflict Rates for the Conflict-Aware and the Eager Scheduler for the Shopping Mix

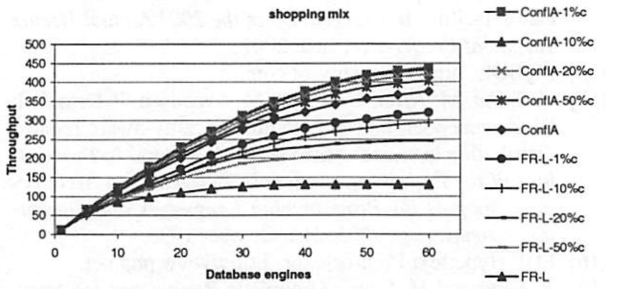


Figure 17: Simulated Throughput Results for Various Conflict Rates for the Conflict-Aware and the FR-L Scheduler for the Shopping Mix

conflict rates as low as 1% of that observed in the experimental workload. Figures 16 and 17 compare throughput as a function of cluster size between the conflict-aware scheduler on one hand, and the eager and the conventional lazy FR-L schedulers on the other hand. We vary the conflict rate from 1% to 100% of the conflict rate observed in the experimental workload. In particular, if a table-level conflict occurs, we ignore the conflict with the specified probability.

Obviously, the performance differences become smaller as the number of conflicts decreases, but at a 1% conflict rate and at the maximum cluster size, the conflict-aware protocol is still a factor of 1.8 better than Eager, and a factor of 1.3 better than a lazy protocol without any optimizations (FR-L). This result demonstrates that conflict-awareness continues to offer benefits for workloads with lower conflict rates or systems with finer-grain concurrency control.

10 Related Work

Current high-volume web servers such as the official web server used for the Olympic games [6] and real-life e-commerce sites based on IBM's WebSphere Commerce Edition [11], rely on expensive supercomputers to satisfy the volume of requests. Nevertheless, performance of such servers may become a problem during periods of peak load.

Neptune [18] adopts a primary-copy approach to providing consistency in a partitioned service cluster. Their scalability study is limited to web applications with loose consistency requirements, such as bulletin boards and auction sites, for which scaling is easier to achieve. They do not address e-commerce workloads or other web applications with stronger consistency requirements.

Zhang et al. [25] have previously attempted to scale dynamic content sites by using clusters in their HACC project. They extend a technique from clustered static-content servers, locality-aware request distribution [15], to work in dynamic content sites. Their study, however, is limited to read-only workloads. In a more general dynamic content server, replication implies the need for consistency maintenance.

Replication has previously been used mainly for fault tolerance and data availability [7]. Gray et al. [8] shows that classic solutions based on eager replication which provide serializability do not scale well. Lazy replication algorithms with asynchronous update propagation used in wide-area applications [17, 21] scale well, but also expose inconsistencies to the user.

More recently, asynchronous replication based on group communication [12, 20, 24] has been proposed to provide serializability and scaling at the same time. This approach is implemented inside the database layer. Each replica functions independently. A transaction acquires locks locally. Prior to commit, the database sends the other replicas the write-sets. Conflicts are solved by each replica independently. This implies the need to abort transactions when a write-set received from a remote transaction conflicts with a local transaction. This approach differs from ours in that we consider a cluster in which a scheduler can direct operations to certain replicas, while they consider a more conventional distributed database setting in which a transaction normally executes locally. Also, our approach is implemented outside of the database.

TP-monitors, such as Tuxedo [9], are superficially similar in functionality to our scheduler. They differ in that they provide programming support for replicated application servers and for accessing different databases using conventional two-phase commit, not transparent support for replicating a database for throughput scaling.

11 Conclusions

We have described conflict-aware scheduling, a lazy replication technique for a cluster of database replicas serving as a back-end to a dynamic content site. A conflict-aware scheduler enforces 1-copy serializability by assigning transaction sequence numbers, and it reduces conflict waiting time by directing reads to replicas where no conflicts exist. This design matches well the characteristics of the database workloads that we have observed in dynamic content sites, namely high locality, high cost of reads relative to the cost of writes, and high conflict rates. No modifications are necessary in the application server or in the database to take advantage of a conflict-aware scheduler.

We have evaluated conflict-aware scheduling, both by measurement of an implementation and by simulation. We use software platforms in common use: the Apache web server, the PHP scripting language, and the MySQL database. We use the various workload mixes of the TPC-W benchmark to evaluate overall scaling behavior and the contribution of our scheduling algorithms. In an 8-node cluster, the conflict-aware scheduler brings factors of 1.5, 1.6 and 2 in throughput improvement, compared to a conflict-oblivious lazy scheduler, and factors of 1.7, 2.4 and 3.5, compared to an eager scheduler, for the browsing, shopping and ordering mixes, respectively.

Furthermore, our simulations show that conflict-aware schedulers scale well to larger clusters and faster machines, and that they maintain an edge over eager and conflict-oblivious schedulers even if the conflict rate is much lower.

Acknowledgments

We would like to thank Gustavo Alonso and Andre Schiper for their advice on early drafts of this paper. We also thank our shepherd, Mike Dahlin, for his guidance, and the anonymous reviewers for their comments. The work also benefited from informal discussions with Emmanuel Cecchet and Karthick Rajamani, and from the contributions of the DynaServer team towards a better understanding of dynamic content benchmarks.

References

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *5th IEEE Workshop on Workload Characterization*, November 2002.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Scaling and availability for dynamic content web sites. Technical Report TR02-395, Rice University, 2002.

[3] Apache Software Foundation. http://www.apache.org/.

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. In *IEEE Transactions on Knowledge and Data Engineering*, volume 2, pages 4–24, March 1990.

[6] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A Publishing System for Efficiently Creating Dynamic Web Data. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[7] R. Flannery. *The Informix Handbook.* Prentice Hall, 2000.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD, June*, pages 173–182, 1996.

[9] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques.* 1993.

[10] N. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation*, February 1999.

[11] A. Jhingran. Anatomy of a real e-commerce system. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data*, May 2000.

[12] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[13] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the 2001 Annual Usenix Technical Conference*, June 2001.

[14] MySQL. http://www.mysql.com.

[15] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, October 1998.

[16] PHP Hypertext Preprocessor. http://www.php.net.

[17] Y. Saito and H. Levy. Optimistic Replication for Internet Data Services. In *International Symposium on Distributed Computing*, October 2000.

[18] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. Kuschner, and H. Zhu. Neptune: Scalable Replica Management and Programming Support for Cluster-based Network Services. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*, pages 207–216, March 2001.

[19] Slashdot: Handling the Loads on 9/11. http://slashdot.org.

[20] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, May 1998.

[21] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings 15th Symposium on Operating Systems Principles*, December 1995.

[22] Transaction Processing Council. http://www.tpc.org/.

[23] G. Weikum and G. Vossen. *Transactional Information Systems. Theory, Algorithms and the Practice of Concurrency Control and Recovery.* Addison-Wesley, Reading, Massachusetts, second edition, 2002.

[24] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, October 2000.

[25] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *Proceedings of the 2000 Annual Usenix Technical Conference*, June 2000.

FastReplica: Efficient Large File Distribution within Content Delivery Networks

Ludmila Cherkasova
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94303
cherkasova@hpl.hp.com

Jangwon Lee
University of Texas at Austin
Austin, Texas 78712
jangwlee@ece.utexas.edu

Abstract. *In this work, we consider a large-scale distributed network of servers and a problem of content distribution across it. We propose a novel algorithm, called FastReplica, for an efficient and reliable replication of large files in the Internet environment. There are a few basic ideas exploited in FastReplica. In order to replicate a large file among n nodes (n is in the range of 10-30 nodes), the original file is partitioned into n subfiles of equal size and each subfile is transferred to a different node in the group. After that, each node propagates its subfile to the remaining nodes in the group. Thus instead of the typical replication of an entire file to n nodes by using n Internet paths, connecting the original node to the replication group, FastReplica exploits $n \times n$ Internet paths within the replication group where each path is used for transferring $\frac{1}{n}$-th of the file. We design a scalable and reliable FastReplica algorithm which can be used for replication of large files to a large group of nodes. The new method is simple and inexpensive. It does not require any changes or modifications to the existing Internet infrastructure, and at the same time, it significantly reduces the file replication time as we demonstrate through experiments on a prototype implementation of FastReplica in a wide-area testbed.*

1 Introduction

Content Delivery Networks (CDNs) are based on a large-scale distributed network of servers located closer to the edges of the Internet for efficient delivery of digital content including various forms of multimedia content. The main goal of the CDN's architecture is to minimize the network impact in the critical path of content delivery as well as to overcome a server overload problem that is a serious threat for busy sites serving popular content.

For typical web documents (e.g. html pages and images) served via CDN, there is no need for active replication of the original content at the edge servers. The CDN's edge servers are the caching servers, and if the requested content is not yet in the cache, this document is retrieved from the original server, using the so-called *pull model.* The performance penalty associated with initial document retrieval from the original server, such as higher latency observed by the client and the additional load experienced by the original server, is not significant for small to medium size web documents.

For large documents, software download packages and media files, a different operational mode is preferred: it is desirable to replicate these files at edge servers in advance, using the so-called *push model.* For large files it is a challenging, resource-intensive problem, e.g. media files can require significant bandwidth and download time due to their large sizes: 20 min media file encoded at 1 Mbit/s results in a file of 150 MBytes.

The sites supported for efficiency reasons by multiple mirror servers, face a similar problem: the original content needs to be replicated across the multiple, geographically distributed, mirror servers.

While transferring a large file with individual point-to-point connections from an original server can be a viable solution in the case of limited number of mirror servers (tenths of servers), this method does not scale when the content needs to be replicated across a CDN with thousands of geographically distributed machines.

Currently, the three most popular methods used for content distribution (replication) in the Internet environment are:

- *satellite distribution,*
- *multicast distribution,*
- *application-level multicast distribution.*

With *satellite distribution* [8, 21], the content distribution server (or the original site) has a transmitting antenna. The servers, to which the content should be replicated, (or the corresponding Internet Data Centers, where the servers are located) have a satellite receiving dish. The original content distribution server broadcasts a file via a satellite channel. Among the shortcomings of the satellite distribution method are that it requires special hardware deployment and the supporting infrastructure (or service) is quite expensive.

With *multicast distribution*, an application can send one copy of each packet and address it to the group of nodes (IP addresses) that want to receive it. This technique reduces network traffic by simultaneously delivering a single stream of information to hundreds/thousands of interested recipients. Multicast can be implemented at both the data-link layer and the network layer. Applications that take advantage of multicast technologies include video conferencing, corporate communications, distance learning, and distribution of software, stock quotes, and news. Among the shortcomings of the multicast distribution method are that it requires a multicast support in routers, which still is not widely available across the Internet infrastructure.

Since the native IP multicast has not received widespread deployment, many industrial and research efforts shifted to investigating and deploying the *application level multicast*, where nodes across the Internet act as intermediate routers to efficiently distribute content along a predefined mesh or tree. A growing number of researchers [7, 9, 12, 13, 17, 10, 6, 14] have advocated this alternative approach, where all multicast related functionality, including group management and packet replication, is implemented at end systems. In this architecture, nodes participating in the multicast group self organize themselves into an scalable overlay structure using a distributed protocol. Further, the nodes attempt to optimize the efficiency of the overlay by adapting to changing network conditions and considering the application level requirements.

An interesting extension for the end-system multicast is introduced in [6], where authors, instead of using the end systems as routers forwarding the packets, propose that the end-systems do actively collaborate in informed manner to improve the performance of large file distribution. The main idea is to overcome the limitation of the traditional service models based on tree topologies where the transfer rate to the client is defined by the bandwidth of the bottleneck link of the path from the server. The authors propose to use additional cross-connections between the end-systems to exchange the complementary content these nodes have already received. Assuming that any given pair of end-systems has not received exactly the same content, these cross-connections between the end-systems can be used to "reconcile" the differences in received content in order to reduce the total transfer time.

In our work, we consider a geographically distributed network of servers and a problem of content distribution across it. Our focus is on distributing large size files such as software packages or stored streaming media files (also called as on-demand streaming media). We propose a novel algorithm, called *FastReplica*, for efficient and reliable replication of large files. There are a few basic ideas exploited in *FastReplica*. In order to replicate a large file among n nodes (n is in the range of 10-30 nodes), the original file is partitioned into n subfiles of equal size and each subfile is transferred to a different node in the group (this way, we introduce a "guaranteed", predictable difference in received content as compared to [6]). After that, each node propagates its subfile to the remaining nodes in the group. Thus instead of the typical replication of an entire file to n nodes by using n Internet paths connecting the original node to the replication group, *FastReplica* exploits $n \times n$ diverse Internet paths within the replication group where each path is used for transferring $\frac{1}{n}$-th of the file (in such a way, similarly to [6], we exploit explicitly the additional cross-connections between the end-systems to exchange the complementary content these nodes have already received).

The paper is organized as follows. Section 2 describes additional related work. In Section 3.2, we introduce a core (an induction step) of the algorithm, called *FastReplica in the small*, and demonstrate its work in the small-scale environment, where a set of nodes in a replication set is rather small and limited, e.g. 10 - 30 nodes. In Section 3.3, we perform a preliminary analysis of *FastReplica in the small* and its potential performance benefits, and outline the configurations and network conditions when *FastReplica* may be inefficient. Then in Section 3.4, using *FastReplica in the small* as the induction step, we design a scalable *FastReplica* algorithm which can be used for replication of large files to a large number of nodes. Finally, in Section 3.5, we show how to extend the algorithm for resilience to nodes' failures.

Through experiments on a prototype implementation, we analyze the performance of *FastReplica in the small* in a wide-area testbed in Section 4. For comparison reasons, we introduce *Multiple Unicast* and *Sequential Unicast* schemas. Under *Multiple Unicast*, the source node simultaneously transfers the entire file to all the recipient nodes via concurrent connections. This schema is traditionally used in the small-scale environment. *Sequential Unicast* schema approximates the file distribution under IP multicast. The experiments show that *FastReplica* significantly reduces the file replication time. On average, it outperforms *Multiple Unicast* by $\frac{n}{2}$ times, where n is the number of nodes in the replication set. Additionally, *FastReplica* demonstrates better or comparable performance against file distribution under *Sequential Unicast*.

While we observed significant performance benefits under *FastReplica* in our experiments, these results are sensitive to a system configuration and bandwidth of the paths between the nodes.

Since *FastReplica in the small* represents an induction step of the general *FastReplica* algorithm, these performance results set the basis for performance expectations of *FastReplica in the large*.

2 Additional Related Work

The recent work on content distribution can be largely divided into three categories: (a) infrastructure based content distribution, (b) overlay network based distribution [7, 9, 12, 13, 17, 10, 6, 14], and (c) peer-to-peer content distribution [11, 16, 1, 24].

Our work is directly related to the infrastructure based content distribution network (CDN) (e.g. Akamai), which employs a dedicated set of machines to reliably and efficiently distribute content to clients on behalf of the server. While the entire collection of nodes in a CDN setting may be varying, we assume that the set of currently active nodes is known. The sites supported by multiple mirror servers are referred to the same category. Existing research on CDNs and server replication has primarily focused on either techniques for efficient redirection of user requests to appropriate servers or content/server placement strategies for reducing the latency of end-users.

A more recent idea is to access multiple servers in parallel to reduce downloading time or to achieve fault tolerance. Several research papers in this direction exploited the benefits of path diversity between the clients and the site's servers with replicated content. Authors in [20], demonstrate the improved response time observed by the client for a large file download through the dynamic parallel access schema to replicated content at mirror servers. Digital Fountain [4] applies Tornado codes to achieve a reliable data download. Their subsequent work [5] reduces the download times by having a client receive a Tornado encoded file from multiple mirror servers. The target application of their approach is bulk data transfer.

While CDNs were originally intended for static web content, they have been applied for delivery of streaming media as well. Delivering streaming media over the Internet is challenging due to a number of factors such as high bit rates, delay and loss sensitivity. Most of the current work in this direction concentrates on how to improve the media delivery from the edge servers (or mirror servers) to the end clients.

In order to improve streaming media quality, the latest work in this direction [3, 15] proposes streaming video from multiple edge servers (or mirror sites), and in particular, by combining the benefits of multiple description coding (MDC) [2] with Internet path diversity. MDC codes a media stream into multiple complementary descriptions. These descriptions have the property that if either description is received it can be used to decode the baseline quality video, and multiple descriptions can be used to decode improved quality video.

One of the basic assumptions in the research papers referred to above is that the original content is already replicated across the edge (mirror) servers. The goal of our paper is to address the content distribution within this infrastructure (and not to the clients of this infrastructure). In this work, we propose a method to efficiently replicate the content (represented by large files) from a single source to a large number of servers in a scalable and reliable way. We exploit ideas of partitioning the original file and using diverse Internet paths between the recipient nodes to speedup the distribution of an original large file over Internet.

In the paper, we partition the original file in n equal subsequent subfiles and apply *FastReplica* to replicate them. This part of the algorithm can be modified accordingly to the nature of the file. For example, for a media file encoded with MDC, different descriptions can be treated as subfiles, and *FastReplica* can be applied to replicate them. Taking into account the nature of MDC (i.e. that either description received by the recipient node can be used to decode the baseline quality video), the part of the *FastReplica* algorithm dealing with nodes failure can be simplified.

3 *FastReplica* Algorithm

In this Section, we describe the formal problem definition and introduce the new algorithm *FastReplica* for replicating the large files in the Internet environment.

In Section 3.2, we introduce a core (an induction step) of the algorithm, called *FastReplica in the small*, and demonstrate its work in the small-scale environment, where a set of nodes to which a file has to be replicated is rather small and limited, e.g. 10 - 30 nodes.

In Section 3.3, we perform a preliminary analysis of *FastReplica in the small* and its potential performance benefits, and outline the configurations and network conditions when *FastReplica* may be inefficient.

In Section 3.4, we describe the general, scalable algorithm, called *FastReplica in the large*, and demonstrate its work in the large-scale environment, where a set of nodes to which a file has to be replicated can be in a range of hundreds/thousands of nodes.

In Section 3.5, we extend the *FastReplica* algorithm to be able to deal with node failures.

3.1 Problem Statement

We use the following notations:

- Let N_0 be a node which has an original file F and let $Size(F)$ denote the size of file F in bytes;
- Let $R = \{N_1, ..., N_n\}$ be a replication set of nodes.

The problem consists in replicating file F across nodes $N_1, ..., N_n$ while minimizing the overall replication time.

3.2 *FastReplica* in the Small

In this Section, we describe a *core* of *FastReplica* which is directly applicable to a case when a set of recipient nodes $N_1, ..., N_n$ is small, e.g. in a range of 10-30 nodes.

File F is divided in n equal subsequent subfiles:

$$F_1, ..., F_n$$

where $Size(F_i) = \frac{Size(F)}{n}$ bytes for each i: $1 \leq i \leq n$.

Step 1: Distribution Step.

The originator node N_0 opens n concurrent network connections to nodes $N_1, ..., N_n$, and sends to each recipient node N_i $(1 \leq i \leq n)$ the following items:

- a distribution list of nodes $R = \{N_1, ..., N_n\}$ to which subfile F_i has to be sent on the next step;
- subfile F_i.

The activities taking place on the first step of the *FastReplica* algorithm are shown in Figure 1. We will denote this step as a *distribution step.*

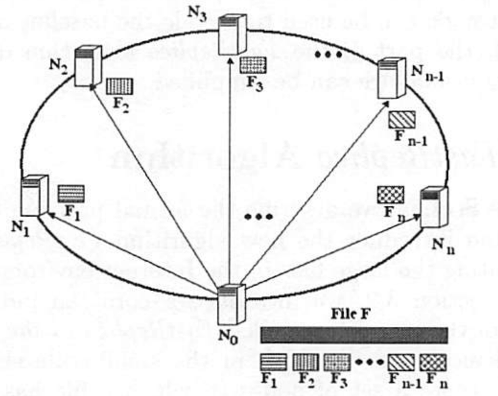


Figure 1: *FastReplica in the small:* distribution step.

Step 2: Collection Step.

After receiving file F_i, node N_i opens $n-1$ concurrent network connections to remaining nodes in the group and send subfile F_i to them as shown in Figure 2 for node N_1.

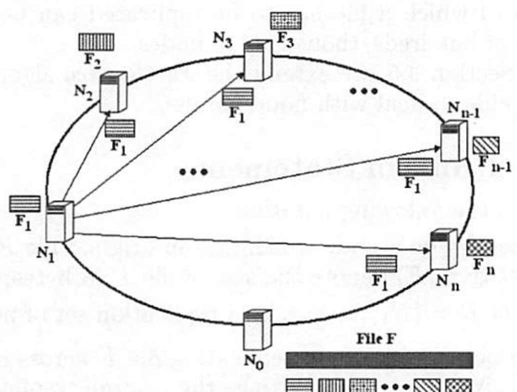


Figure 2: *FastReplica in the small:* a set of outgoing connections of node N_1 at collection step.

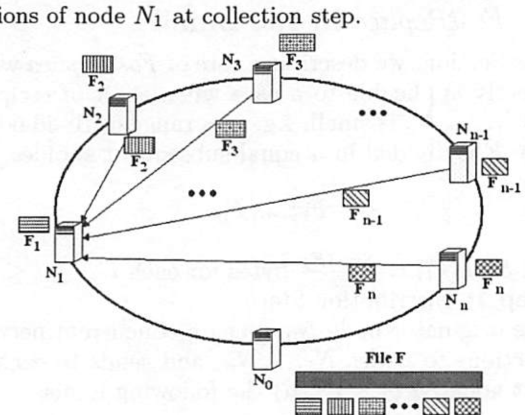


Figure 3: *FastReplica in the small:* a set of incoming connections of node N_1 at collection step.

Similarly, Figure 3 shows the set of incoming concurrent connections to node N_1 from the remaining nodes $N_2, ..., N_n$ transferring the complementary subfiles $F_2, ..., F_n$ during the second logical step of the algorithm. Thus at this step, each node N_i has the following set of network connections:

- there are $n-1$ outgoing connections from node N_i: one connection to each node N_k ($k \neq i$) for sending the corresponding subfile F_i to node N_k.
- there are $n-1$ incoming connections to node N_i: one connection from each node N_k ($k \neq i$) for sending the corresponding subfile F_k to node N_i.

Thus at the end of this step, each node receives all subfiles $F_1, ..., F_n$ comprising the entire original file F. We will denote this step as a *collection step.*

In summary, the main idea behind *FastReplica* is that instead of the typical replication of an entire file to n nodes by using n Internet paths connecting the original node to the replication group, the *FastReplica* algorithm exploits $n \times n$ different Internet paths within the replication group where each path is used for transferring $\frac{1}{n}$-th of the file. Thus, the impact of congestion on any particular Internet path participating in the schema is limited for a transfer of $\frac{1}{n}$-th of the file. Additionally, *FastReplica* takes advantage of the upload and download bandwidth of the recipient nodes.

3.3 Preliminary Performance Analysis of *FastReplica* in the Small

Let $Time^i(F)$ denote the transfer time of file F from the original node N_0 to node N_i as measured at node N_i. We use *transfer time* and *replication time* interchangeably in the text. In our study, we consider the following two performance metrics:

- *Average replication time:*

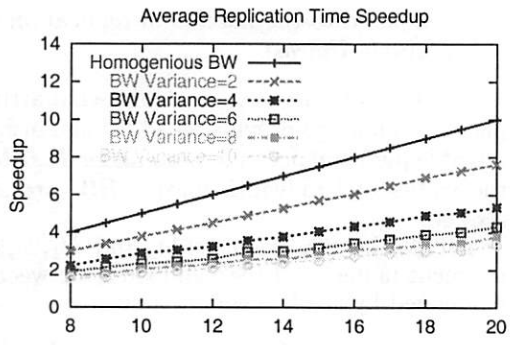
$$Time_{aver} = \frac{1}{n}\sum_{i=1}^{n} Time^i(F)$$

- *Maximum replication time:*

$$Time_{max} = max\{Time^i(F)\}, i \in \{1, \cdots, n\}$$

$Time_{max}$ reflects the time when all the nodes in the replication set receive a copy of the original file, and the primary goal of *FastReplica* is to minimize the maximum replication time. However, we are also interested in understanding the impact of *FastReplica* on the average replication time $Time_{aver}$.

First, let us consider an *idealistic setting,* where nodes $N_1, ..., N_n$ have symmetrical (or nearly symmetrical) incoming and outgoing bandwidth which is typical for CDNs, distributed IDCs, and a distributed enterprise environment. In addition, let nodes $N_0, N_1, ..., N_n$ be homogeneous, and let each node can support k network connections to other nodes at B bytes per second on average.

a) 

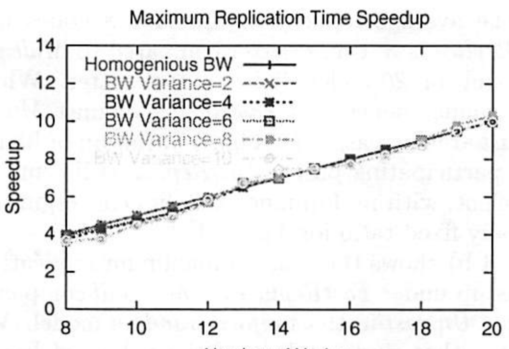
b) 

Figure 4: *Uniform-random model*: speedup in file replication time under *FastReplica* vs *Multiple Unicast* for a different number of nodes in replication set for a) average replication time, and b) maximum replication time.

In the idealistic setting, there is no difference between maximum and average replication times. Using the assumption on homogeneity of nodes' bandwidth, we can estimate the transfer time for each concurrent connection i ($1 \le i \le n$) during the *distribution step*:

$$Time_{distr} = \frac{Size(F)}{n \times B} \quad (1)$$

The transfer time at the *collection step* is similar to the time encountered at the first (distribution) step:

$$Time_{collect} = \frac{Size(F)}{n \times B} \quad (2)$$

Thus the overall replication time under *FastReplica in the small* is the following:

$$Time_{FR}^{small} = Time_{distr} + Time_{collect} = 2 \times \frac{Size(F)}{n \times B} \quad (3)$$

Let *Multiple Unicast* denote a schema that transfers the entire file F from the original node N_0 to nodes $N_1, ..., N_n$ by simultaneously using n concurrent network connections. The overall transfer time under *Multiple Unicast* is the following:

$$Time_{MU}^{small} = \frac{Size(F)}{B} \quad (4)$$

Thus, in an idealistic setting, *FastReplica in the small* provides the following speedup of file replication time compared to the *Multiple Unicast* strategy:

$$Replication\_Time\_Speedup = \frac{Time_{FR}^{small}}{Time_{MU}^{small}} = \frac{n}{2} \quad (5)$$

While the comparison of *FastReplica* and *Multiple Unicast* in the idealistic environment gives insights into why the new algorithm may provide significant performance benefits for replication of the large files, the bandwidth conditions in the realistic setting could be very different from the idealistic assumptions. Due to changing network conditions, even the same link might have a different available bandwidth when measured at different times. Let us analyze how *FastReplica* performs when network paths participating in the transfers have a different available bandwidth.

Let BW denote a *bandwidth matrix*, where $BW[i][j]$ reflects the available bandwidth of the path from node N_i to node N_j as measured at some time T, and let *Var* be the ratio of maximum to minimum available bandwidth along the paths participating in the file transfers. We call *Var* a *bandwidth variation.*

In our analysis, we consider the bandwidth matrix BW to be populated in the following way:

$$BW[i][j] = B \times random(1, Var),$$

where function $random(1, Var)$ returns a random integer var: $1 \le var \le Var$.

While it is still a simplistic model, it helps to reflect a realistic situation, where the available bandwidth of different links can be significantly different. We will call this model a *uniform-random model.* To perform a sensitivity analysis of how the *FastReplica* performance depends on a bandwidth variation of participating paths, we experimented with a range of different values for *Var* between 1 and 10. When $Var = 1$, it is the *idealistic setting*, discussed above, where all of the paths are homogeneous and have the same bandwidth B (i.e. no variation in bandwidth). When $Var = 10$, the network paths between the nodes have highly variable available bandwidth with a possible difference of up to 10 times.

Using the *uniform-random* model and its bandwidth matrix BW, we compute the average and maximum file replication times under *FastReplica* and *Multiple Unicast* methods for a different number of nodes in the replication set, and derive the relative speedup of the file replication time under *FastReplica* compared to the replication time under the *Multiple Unicast* strategy. For each value of *Var*, we repeated the experiments multiple times, where the bandwidth matrix BW is populated by using the random number generator with different seeds.

Figure 4 a) shows the relative average replication time speedup under *FastReplica in the small* compared to *Multiple Unicast* in the *uniform-random* model. For

Var=2, the average replication time for *8* nodes under *FastReplica* is *3* times better compared to *Multiple Unicast*, and for *20* nodes, it is *8* times better. While the performance benefits of *FastReplica* against *Multiple Unicast* are decreasing for higher variation of bandwidth of participating paths, *FastReplica* still remains quite efficient, with performance benefits converging to a practically fixed ratio for $Var > 4$.

Figure 4 b) shows the relative maximum replication time speedup under *FastReplica in the small* compared to *Multiple Unicast* in the *uniform-random* model. We can observe that, independent of the values of bandwidth variation, the maximum replication time under *FastReplica* for n nodes is $\frac{n}{2}$ times better compared to the maximum replication time under *Multiple Unicast*.

It can be explained in the following way:

- *Multiple Unicast*: The maximum replication time is defined by the entire file transfer time over the path with the worst available bandwidth among the paths connecting N_0 and N_i, $1 \leq i \leq n$.
- *FastReplica*: Figure 5 shows the set of paths participating in the file transfer from node N_0 to node N_1 under the *FastReplica* algorithm (we use N_1 as a representative of the recipient nodes).

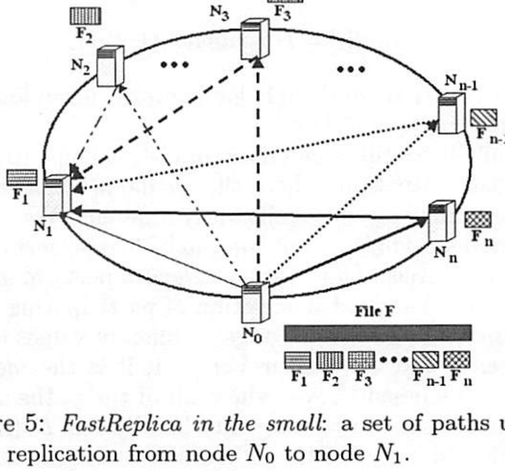


Figure 5: *FastReplica in the small*: a set of paths used in file F replication from node N_0 to node N_1.

The replication time observed at node N_1 is defined by the maximum transfer time of $\frac{1}{n}$-th of the file over either:

- the path from N_0 to N_1, or
- the path with the worst overall available bandwidth consisting of two subpaths:
  - the subpath from N_0 to N_j and
  - the subpath from N_j to N_1,

  for some $j : 1 \leq j \leq n$.

  In the considered *uniform-random* model, a worst case scenario is when both subpaths have a minimal bandwidth, and since each path is used for transferring $\frac{1}{n}$-th of the entire file, this would lead to $\frac{n}{2}$ times latency improvement under *FastReplica* compared to the maximum replication time under *Multiple Unicast*.

Now, let us consider a special, somewhat artificial example, which aims to provide an additional insight into the possible performance outcomes under *FastReplica* when the values of bandwidth matrix BW are significantly skewed.

Let N_0 be the origin node, and $N_1, ..., N_{10}$ be the recipient nodes, and the bandwidth between the nodes be defined by the following matrix:

$$BW(i,j) = \begin{cases} \frac{1}{10} \times B & \text{if i=0, j=1} \\ B & \text{if i=0, } 2 \leq j \leq 10 \\ \frac{1}{10} \times B & \text{if } 1 \leq i,j \leq 10 \end{cases} \quad (6)$$

In other words, the origin node N_0 has a limited bandwidth of $\frac{1}{10} \times B$ to node N_1, while the bandwidth from N_0 to the rest of the recipient nodes $N_2, ..., N_{10}$ is equal to B. In addition, the cross-bandwidth between the nodes $N_1, ..., N_{10}$ is also very limited, such that any pair N_i and N_j is connected via a path with available bandwidth of $\frac{1}{10} \times B$.

At a glance, it seems that *FastReplica* might perform badly in this configuration because the additional cross-bandwidth between the recipient nodes $N_1, ..., N_{10}$ is so poor relative to the bandwidth available between the origin node N_0 and the recipient nodes $N_2, ..., N_{10}$. Let us compute the average and maximum replication times for this configuration under *Multiple Unicast* and *FastReplica* strategies.

- *Multiple Unicast*:

$$Time_{aver} = \frac{19 \times Size(F)}{10 \times B}, Time_{max} = \frac{10 \times Size(F)}{B}.$$

- *FastReplica*:

$$Time_{aver} = \frac{191 \times Size(F)}{100 \times B}, Time_{max} = \frac{2 \times Size(F)}{B}.$$

The maximum replication time in this configuration is 5 times better under *FastReplica* than under *Multiple Unicast*. In *FastReplica*, any path between the nodes is used to transfer only $\frac{1}{n}$-th of the entire file. Thus, the paths with poor bandwidth are used for much shorter transfers which leads to a significant improvement in maximum replication time. However, the average replication time in this example is not improved under *FastReplica* compared to *Multiple Unicast*. The reason for this is that the high bandwidth paths in this configuration are used similarly: to transfer only $\frac{1}{n}$-th of the entire file, and during the collection step of the *FastReplica* algorithm, the transfers of complementary $\frac{1}{n}$-th size subfiles within the replication group are performed over poor bandwidth paths. Thus, in certain cases, like considered above, *FastReplica* may provide significant improvements in maximum replication time, but may not improve the average replication time.

The analysis considered in this section outlines the conditions when *FastReplica* is expected to perform well, providing the essential performance benefits. Similar reasoning can be applied to derive the situations when *FastReplica* might be inefficient. For example, let us slightly modify the previous example. Let the bandwidth matrix BW be defined in the following way:

$$BW(i,j) = \begin{cases} B & \text{if i=0, } 1 \leq j \leq 10 \\ \frac{1}{10} \times B & \text{if } 1 \leq i,j \leq 10 \end{cases} \qquad (7)$$

In this configuration, the bandwidth from the origin node N_0 to the rest of the recipient nodes $N_1, ..., N_{10}$ is equal to B, while the cross-bandwidth between the nodes $N_1, ..., N_{10}$ is very limited: any pair N_i and N_j is connected via a path with available bandwidth of $\frac{1}{10} \times B$. The average and maximum replication times for this configuration under *Multiple Unicast* and *FastReplica* strategies can be computed as follows:

- *Multiple Unicast*: $Time_{aver} = Time_{max} = \frac{Size(F)}{B}$.
- *FastReplica*: $Time_{aver} = Time_{max} = \frac{11 \times Size(F)}{10 \times B}$.

Thus in this configuration, *FastReplica* does not provide any performance benefits.

In a general case, if there is a node N_k in the replication set such that most of the paths between N_k and the rest of the nodes have a very limited available bandwidth (say, n times worse than the minimal available bandwidth of the paths connecting N_0 and N_i, $1 \leq i \leq n$) then the performance of *FastReplica* during the second (collection) step is impacted by the poor bandwidth of the paths between N_k and N_i, $1 \leq i \leq n$, and *FastReplica* will not provide expected performance benefits. Note, that n (the number of nodes in the replication group) plays a very important role here: a larger value of n provides a higher "safety" level for *FastReplica* efficiency. A larger value of n helps to offset a higher difference in bandwidth between the available bandwidth within the replication group and the available bandwidth from the original node to the nodes in the replication group.

To apply *FastReplica* efficiently, the preliminary bandwidth estimates are useful. These bandwidth estimates are also essential for correct clustering of the appropriate nodes into the replication subgroups in *FastReplica in the large* discussed in the next section.

3.4 *FastReplica* in the Large

In this Section, we generalize *FastReplica in the small* to a case where a set of nodes to which a file has to be replicated can be in the range of hundreds/thousands of nodes.

Let k be a number of network connections chosen for concurrent transfers between a single node and multiple receiving nodes (i.e. k limits the number of nodes in the group for *Multiple Unicast* or *FastReplica* strategies). An appropriate value of k can be experimentally determined via probing. Heterogeneous nodes might be capable of supporting a different number of connections. Let k be the number of connections suitable for most of the nodes in the overall replication set.

A natural way to scale *FastReplica in the small* to a large number of nodes is:

- partition the original set of nodes into replication groups, each consisting of k nodes;
- apply *FastReplica in the small* iteratively: first, replicate the original file F to a group of k nodes, and then use these k nodes as the origin nodes with file F to repeat the same procedure to a new groups of nodes, etc.

Schematically, this procedure is shown in Figure 6, where circles represent the nodes, and boxes represent the replication groups. The arrows, connecting one node with a set of other nodes, reflect the origin node and the recipient nodes, involved in communications on a particular iteration of the algorithm.

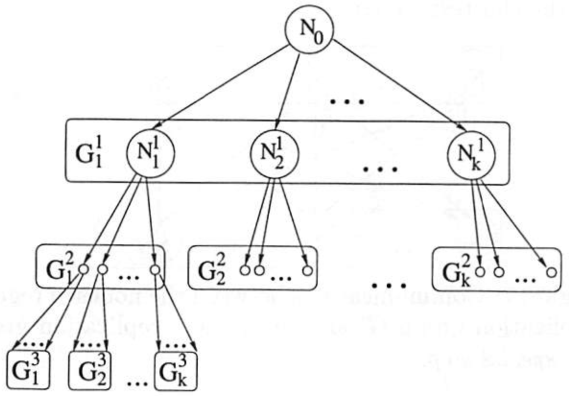


Figure 6: *FastReplica in the large*: iterative replication process.

At the first iteration, node N_0 replicates file F to group G_1^1, consisting of k nodes, by using the *FastReplica in the small* algorithm.

At the second iteration, each node $N_i^1 (1 \leq i \leq k)$ of group G_1^1 can serve as the origin node propagating file F to another group G_i^2.

Thus in two iterations, file F can be replicated to $k \times k$ nodes. Correspondingly, in three iterations, file F can be replicated to $k \times k \times k$ nodes.

The general *FastReplica* algorithm is based on the reasoning described above. Let the problem consist in replicating file F across nodes $N_1, ..., N_n$ and let $\frac{n}{k} = m$. Then all the nodes are partitioned into m groups:

$$G^1, G^2, ..., G^m$$

where each group has k nodes.

Any number m can be represented as

$$m = c_1 \times k^{i_1} + c_2 \times k^{i_2} + ... + c_j \times k^{i_j} \qquad (8)$$

where $i_1 > i_2 > ... > i_j \geq 0$ and $0 < c_1, ..., c_j < k$. Practically, it is a k-ary representation of a number m.

This representation defines the rules for constructing the tree structure similar to the one shown in Figure 6. In particular, the height of such a tree is i_1+1, and it defines the number of iterations in the general *FastReplica* algorithm.

From this representation, the rules for constructing the corresponding distribution lists of nodes are straightforward. We omit the technical details of the distribution lists construction in order to keep the description of the overall algorithm concise.

If the targeted number n of nodes for a file replication is not a multiple of k, i.e.

$$\frac{n}{k} = m + r$$

where $r < k$, then there is one "incomplete" group $\hat{G}$ with r nodes in it. The best way to deal with this group is to arrange it to be a leaf-group in the shortest subtree. Let $G' = \{N'_1, ..., N'_k\}$ be a replication group in the shortest subtree.

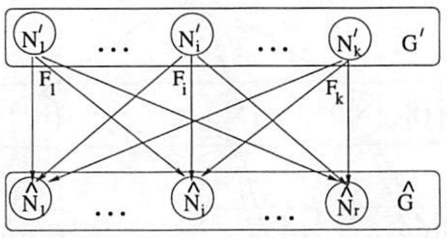


Figure 7: Communications between the nodes of regular replication group G' and incomplete replication group $\hat{G}$: *special step.*

The communications between groups G' and $\hat{G}$ follow a slightly different file exchange protocol. All the nodes in G' have already received all subfiles $F_1, ..., F_n$ comprising the entire original file F. Each node N'_i of group G' opens r concurrent network connections to all r nodes of group $\hat{G}$ for transferring its subfile F_i as shown in Figure 7. In this way, at the end of this step, each node of group $\hat{G}$ has all subfiles $F_1, ..., F_k$ of the original file F. We will denote this step as a *special step.*

Example. Let $k = 10$. How many algorithm iterations are required to replicate the original file to 1000 nodes? Using Formula (8) above, we derive the following representation for 1000 nodes:

$$1000 = 10 \times 10^2$$

Thus, in three algorithm iterations ($10 \times 10 \times 10$), the original file can be replicated among all 1000 nodes. At each iteration, the replication process follows *FastReplica in the small,* i.e. the iteration consists of 2 steps, each used for transferring the $\frac{1}{k}$-th portion of the original file F.

Let *Multiple Unicast* follow a similar recursive replication tree as the one defined above in general *FastReplica* and shown in Figure 6, with the only difference being that communications between the origin nodes and the recipient nodes follow the *Multiple Unicast* schema, i.e. the origin node transfers the entire file F to the corresponding recipient nodes by simultaneously using k concurrent network connections. Thus, in three algorithm iterations, by using *Multiple Unicast* recursively, the original file can be replicated among 1000 nodes.

3.5 Reliable *FastReplica* Algorithm

In this Section, we extend the *FastReplica* algorithm to be able to deal with node failures. The basic algorithm presented in Sections 3.2, 3.4 is sensitive to node failures. For example, if node N_1 fails during either transfer shown in Figures 1, 2 then this event may impact all nodes $N_2, ..., N_n$ in the group because each node depends on node N_1 to receive subfile F_1. In the described scenario, node N_1 is acting as a recipient node in the replication set. If a node fails when it acts as the origin node, e.g. node N_1^1 in Figure 6, this failure impacts all of the replication groups in the replication subtree rooted in node N_1^1.

The reliable *FastReplica* algorithm proposed below efficiently deals with node failures by making the local repair decision within the particular group of nodes. It keeps the main structure of the *FastReplica* algorithm practically unchanged while adding the desired property of resilience to node failures.

In reliable *FastReplica*, the nodes of each group are exchanging the heartbeat messages with their origin node. The heartbeat messages from nodes to their origin node are augmented with additional information on the corresponding algorithm step and group (list) of nodes to which the nodes currently perform their transfers.

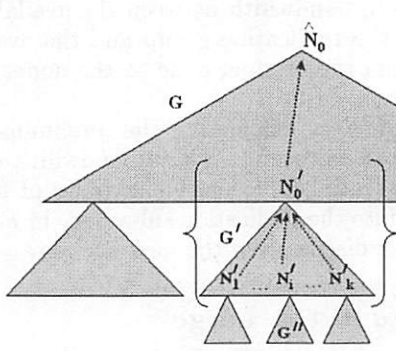


Figure 8: Heartbeat group: the recipient nodes in $G' = \{N'_1, ..., N'_k\}$ send heartbeat messages to the origin node N'_0.

In Figure 8, the nodes $N'_1, ..., N'_k$ of group G' form the heartbeat group with their origin node N'_0. Each node N'_i sends to N'_0 the heartbeat messages with additional information on node state in the replication process. Similarly, node N'_0 belongs to group G with the corresponding origin node $\hat{N}_0$. Thus node N'_0 sends the heartbeat messages and its node state to $\hat{N}_0$.

There are different repair procedures depending on whether a failed node was acting as a recipient node, e.g. node N_i' in replication set G', or a failed node was acting as an origin node, e.g. N_0' for replication set G'.

- If node N_i' fails while acting as a recipient node in replication set G' during the *distribution step* then the communication pattern is similar to the pattern shown in Figure 1. In this case, node N_0' is aware of the node N_i' failure. Node N_0' performs the following repair step: it uses $k-1$ already opened connections to the rest of the nodes in group G' to send the missing F_i file to each node in the group as shown in Figure 9.

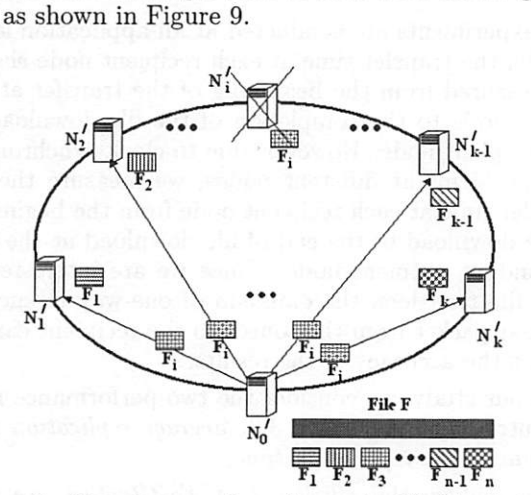


Figure 9: Repair procedure for node N_i' failed during *distribution step.*

In this way, each node in group G' receives all of the subfiles of the original file F.

Additionally, node N_0' acts as a "substitute" for the failed node N_i' in the next algorithm step. If node N_i' was supposed to serve as the origin node to group G'' for the next algorithm iteration, then node N_0' acts as the origin node to group G'' for this iteration.

- If node N_i' fails while acting as a recipient node in replication set G' during the *collection step* then the communication pattern is similar to the pattern shown in Figure 2. Using the heartbeat messages, the failure of node N_i' is detected by node N_0'. Node N_0' performs the following repair step: it opens connections to the impacted nodes in group G' to send missing file F_i (similar to the repair step shown in Figure 9). In this way, each node in group G' receives all of the subfiles of the original file F.

  Analogously, node N_0' acts as a substitute for the failed node N_i' in the next algorithm step.

- If node N_0' fails while acting as the origin node for replication group G' during the distribution step then replication group G' should be "reattached" to a higher-level origin node. Let $\hat{N}_0$ be the corresponding origin node for N_0' from the previous iteration step as shown in Figure 8. From heartbeat messages, node $\hat{N}_0$ detects node N_i' failure. Node $\hat{N}_0$ analyzes what was the node N_0' state in the replication process preceding its failure. Then node $\hat{N}_0$ acts as a replacement for N_0': it opens connections to the impacted nodes in group G' to send corresponding missing files. Additionally, $\hat{N}_0$ updates every node in G' about the change of the origin node (for future exchange of heartbeat messages).

Reliable *FastReplica*, described above, aims to minimize the impact of node failures by making the local repair decision within the particular group of nodes. These groups are relatively small, e.g. 10-30 nodes. Each group has the origin node (with the original file for replication) and the recipient nodes. The number of heart-bit messages in such a group is very small because only the recipient nodes send heart-bit messages to their origin node, and there are no heart-bit messages between the recipient nodes. This structure significantly simplifies the protocol. Proposed failure mechanism easily handles a single node failure within the group with minimal performance penalty. The main structure of the *FastReplica* algorithm is practically unchanged during the repair steps.

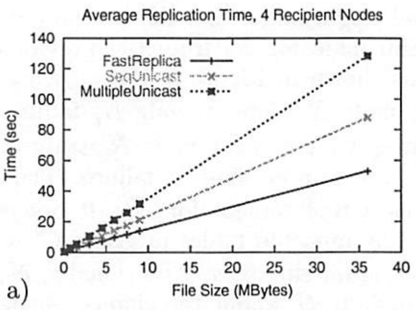
4 Performance Evaluation

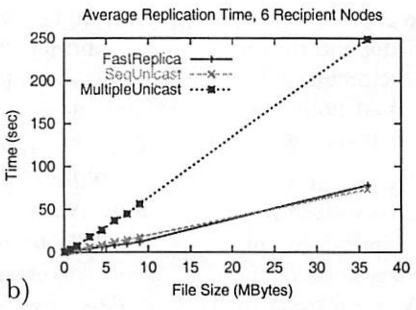
We outlined the potential performance benefits of *FastReplica in the small* in Section 3.3. The goal of this section is to analyze the *FastReplica* performance in the real Internet environment. Through experiments on a prototype implementation, we will demonstrate the efficiency of *FastReplica in the small* in a wide-area testbed. Since *FastReplica in the small* defines the iteration step in the general algorithm, these results will set the basis for performance expectations of *FastReplica in the large.*

Using the generous help of summer interns at HPLabs, we built an experimental testbed with 9 nodes. Table 1 and Figure 10 show the 9 hosts participating in our experiments and their geographic locations.

N_0	hp.com	Palo Alto, CA
N_1	utexas.edu	Austin, TX
N_2	umich.edu	Ann Arbor, MI
N_3	gatech.edu	Atlanta, GA
N_4	duke.edu	Durham, NC
N_5	uci.edu	Irvine, CA
N_6	berkeley.edu	Berkeley, CA
N_7	mit.edu	Cambridge, MA
N_8	uiuc.edu	Urbana-Champaign, IL

Table 1: Participating nodes.

a) 

b) 

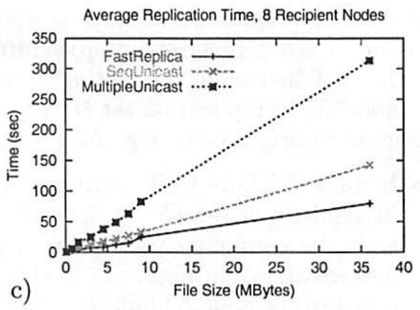
c) 

Figure 11: Average replication time for files of different size and a different number of nodes in replication set a) 4 receivers, b) 6 receivers, c) 8 receivers.

The source node is N_0 and is located at the HP site, while the nodes-receivers are at different university sites. In order to perform the sensitivity analysis, we vary the number of participating hosts: in experiments with k participating hosts in replication set, the receivers are $N_1, ..., N_k$ ordered as shown in Table 1.

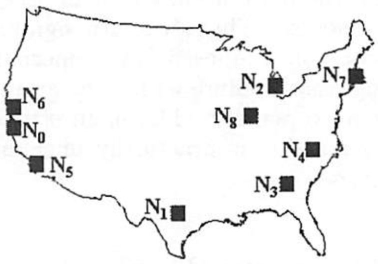


Figure 10: Geographic locations of hosts.

Using the experimental testbed, we compare the following distribution schemes:

- *FastReplica in the small:* we used the *FastReplica* algorithm designed for small, limited number of nodes and introduced in Section 3.2.
- *Sequential Unicast:* this scheme approximates the "best possible overlay tree" for the entire set of group members. For the evaluation, we use the *Sequential Unicast test* which measures the file transfer time from the source to each recipient *independently* (i.e. in the absence of other recipients). Note that *Sequential Unicast* is not a feasible overlay, but a hypothetical construction used for comparison purposes. The measurements under *Sequential Unicast* approximate the file distribution using IP multicast.
- *Multiple Unicast:* under this scheme, the original node simultaneously transfers the entire file to all the recipient nodes by using the concurrent connections. Assuming an infinite bandwidth at the original node, this scheme can be considered as a feasible solution for the above "best possible overlay tree".

The experiments are conducted at an application level. Ideally, the transfer time at each recipient node should be measured from the beginning of the transfer at the source node to the completion of the file download at the recipient node. However, due to clock synchronization problems at different nodes, we measure the file transfer time at each recipient node from the beginning of file download to the end of file download at the corresponding recipient node. Since we are interested in large file transfers, the omission of one-way latency of the first packet from the source to the recipient cannot impact the accuracy of the results.

In our study, we consider the two performance metrics introduced in Section 3.3: *average replication time* and *maximum replication time*.

To analyze the efficiency of *FastReplica*, we performed its sensitivity analysis for replication of different size files and across different numbers of nodes in the replication set. We experimented with 9 file sizes: 80 Kbytes, 750 Kbytes, 1.5 MBytes, 3 MBytes, 4.5 MBytes, 6 MBytes, 7.5 MBytes, 9 MBytes, and 36 MBytes, and varied the number of nodes in the replication set from 2 to 8. When running experiments with different parameters and strategies, the experiments for the same file size were clustered in time as closely as possible to eliminate biases due to short time scale changes in network and system conditions. In order to eliminate the biases due to longer time scale changes in network and system conditions, we performed the same set of experiments at different times of the day. Each point in the results is averaged over 10 different runs which were performed over a 10 day period.

Figure 11 shows the average file replication time for experiments with 4, 6, and 8 recipient nodes in the replication set and files of different sizes. For file sizes larger than 80 Kbytes, *FastReplica* significantly outperforms *Multiple Unicast*. The replication time under *FastReplica* is 2-4 times better than under *Multiple Unicast*.

Additionally, in most experiments, *FastReplica* outperforms *Sequential Unicast*, which approximates the file replication with IP multicast. The explanation is that when *Sequential Unicast* replicates file F to n nodes, it uses n Internet paths connecting the source

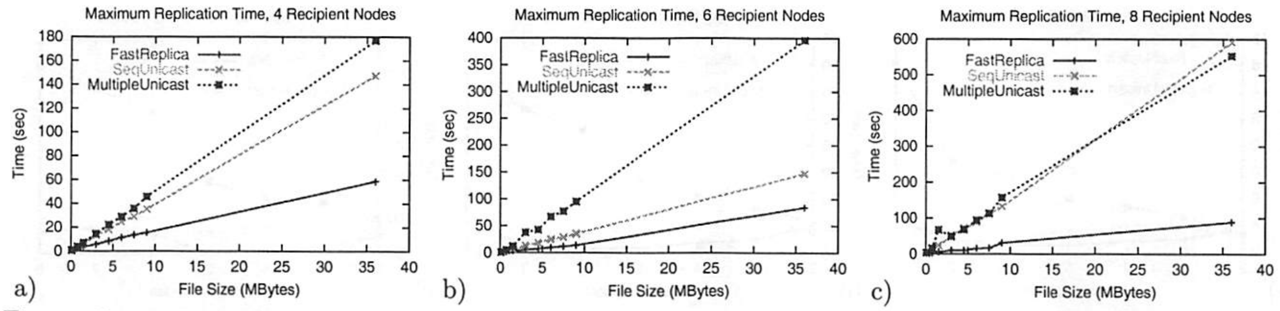


Figure 12: Maximum replication time for files of different size and a different number of nodes in replication set a) 4 receivers, b) 6 receivers, c) 8 receivers.

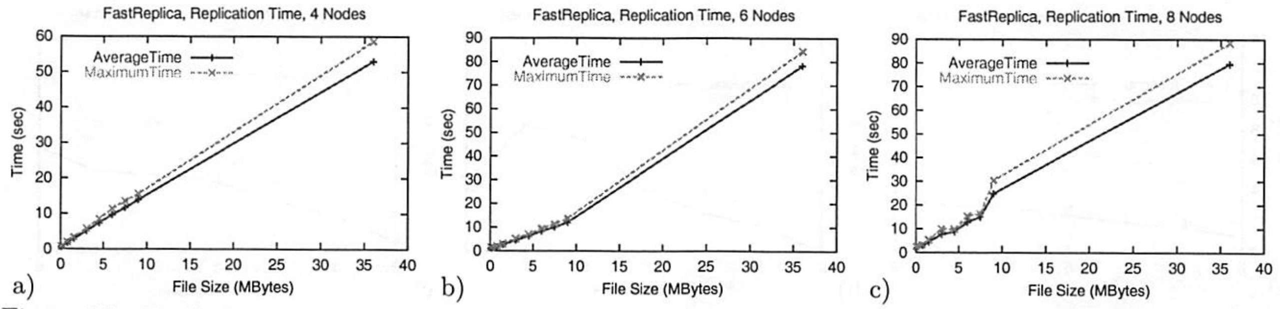


Figure 13: *FastReplica*: average vs maximum replication time for a different number of nodes in replication set a) 4 receivers, b) 6 receivers, c) 8 receivers.

nodes to the recipient nodes (while sending only one packet over each common link in those paths). Thus the overall performance is defined by the end-to-end properties of the n paths. Congestion in any of those paths impacts the overall performance of the *Sequential Unicast*. *FastReplica* uses the same n paths between the source and recipient nodes to transfer only $\frac{1}{n}$-th of file F. *FastReplica* takes advantage of using the additional $(n-1) \times n$ paths between the nodes in the replication set, and each of those paths is used for sending $\frac{1}{n}$-th of file F. Thus, the congestion in any of those paths impacts *FastReplica* performance for transfer of only the $\frac{1}{n}$-th of file F.

While the average replication time provides an interesting metric for distribution strategy characterization, the metric representing the maximum replication time is critical, because it reflects the worst case of the replication time among the recipient nodes. Figure 12 shows the maximum replication time for experiments with 4, 6, and 8 recipient nodes in a replication set and files of different sizes. The maximum replication times under *Multiple Unicast*, as well as *Sequential Unicast*, are much higher than the corresponding average times for these strategies. For a case of 8 nodes in the replication set, the maximum times under *Multiple Unicast* and *Sequential Unicast* are almost 2 times higher than the corresponding average times. The reason is that there is a very limited bandwidth on the path from the source node N_0 to the recipient node N_8. The performance of this path is practically the same for both *Multiple Unicast* and *Sequential Unicast*. This path defines the worst (maximum) replication time among all the recipient nodes in the set. Since *FastReplica* uses this path to transfer only $\frac{1}{n}$-th of file F, this "bad" path has a very limited impact on maximum replication time and overall performance of *FastReplica*.

Figure 13 shows how close the average and maximum replication times under *FastReplica* are. These results demonstrate the robustness and predictability of performance results under the new strategy.

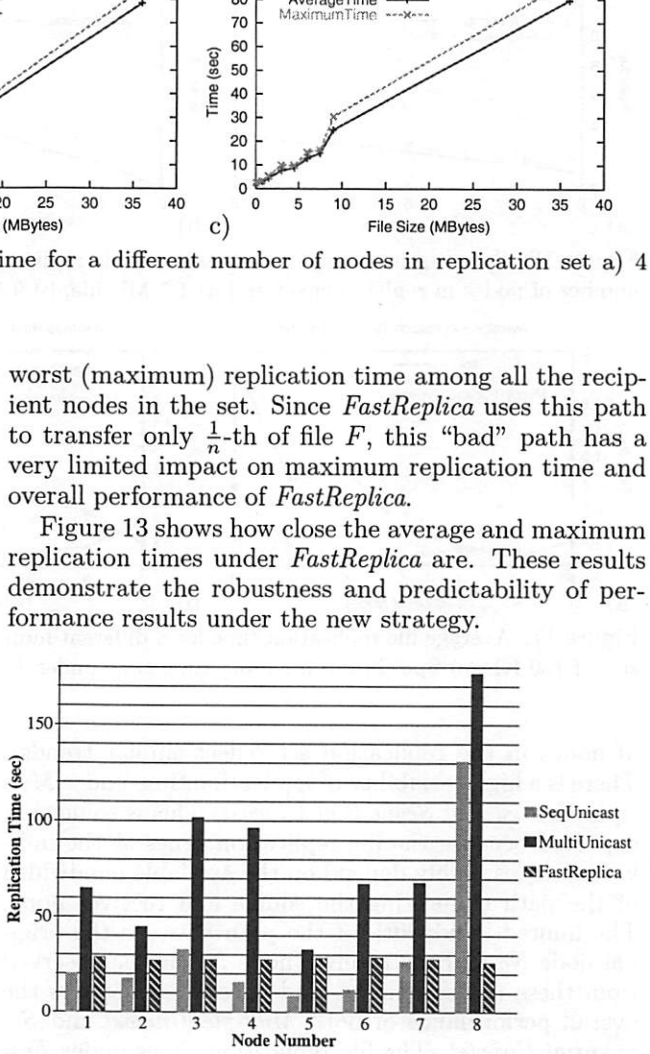


Figure 14: Replication time measured by individual receiving nodes for 9 MB file and 8 nodes in replication set.

Figure 14 shows the average replication time measured by the different, individual recipient nodes for a 9 MB file and 8 nodes in the replication set (the other graphs for different file sizes and a different number

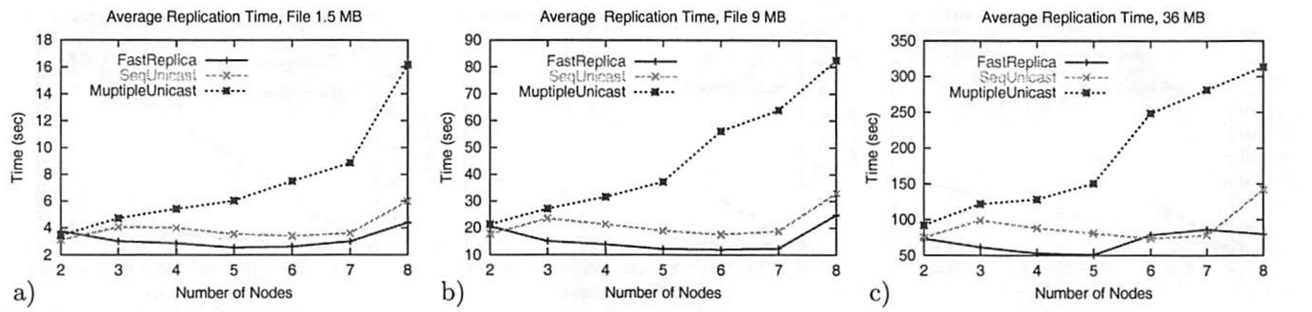


Figure 15: Average file replication time for a different number of nodes in replication set and a) File size of 1.5 MB, b) File size of 9 MB, c) File size of 36 MB.

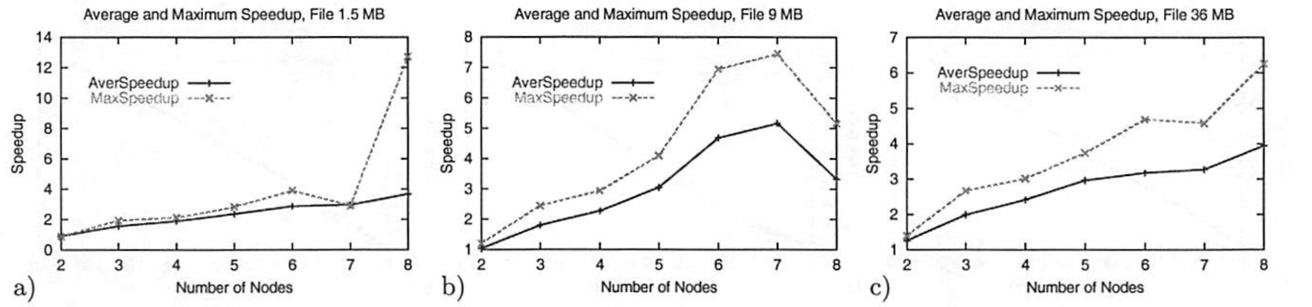


Figure 16: Speedup in average and maximum file replication time under *FastReplica* vs *Multiple Unicast* for a different number of nodes in replication set and a) 1.5 MB file, b) 9 MB file, c) 36 MB file.

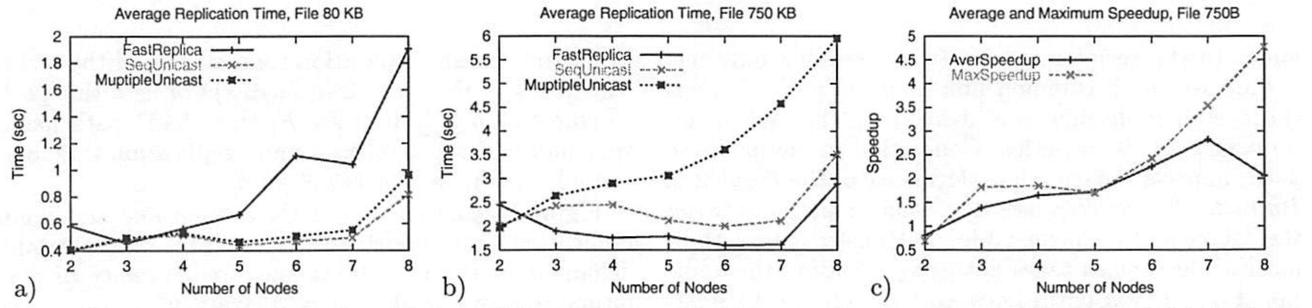


Figure 17: Average file replication time for a different number of nodes in replication set and a) File size of 80 KB, b) File size of 750 KB, c) Speedup in file replication time under *FastReplica* vs *Multiple Unicast* for 750 KB file.

of nodes in the replication set reflect similar trends). There is a high variability of replication time under *Multiple Unicast* and *Sequential Unicast*. This is somewhat expected because the file replication times at the individual nodes highly depend on the available bandwidth of the path connecting the source and receiver node. The limited bandwidth of the path between the original node N_0 and the receiver node N_8 can be observed from these measurements, and it severely impacts the overall performance of both *Multiple Unicast* and *Sequential Unicast*. The file replication times under *FastReplica* across different nodes in the replication set are much more stable and predictable since each node performance is defined by the bandwidth of n paths, each transferring $\frac{1}{n}$-th of the original file F.

Figure 15 shows the average replication time for files of 1.5 MB, 9 MB, and 36 MB for a different number of nodes in the replication set. While *Multiple Unicast* shows a growing replication time for an increasing number of nodes in the replication set, *FastReplica* and *Sequential Unicast* demonstrate good scalability for replication sets of different sizes. Additionally, *FastReplica* consistently outperforms *Sequential Unicast* for most of the points.

Figure 16 shows the average and maximum speedup of file replication time under proposed *FastReplica in the small* relative to the replication time of *Multiple Unicast* for files of 1.5 MB, 9 MB, and 36 MB, and a different number of nodes in the replication set. The results consistently show the significant speedup both in average and maximum replication times across considered different file sizes.

Finally, Figures 17 a) and b) show the average replication time for 80 KB and 750 KB files and a different number of nodes in the replication set. The files of 80 KB and 750 KB are the smallest ones used in our

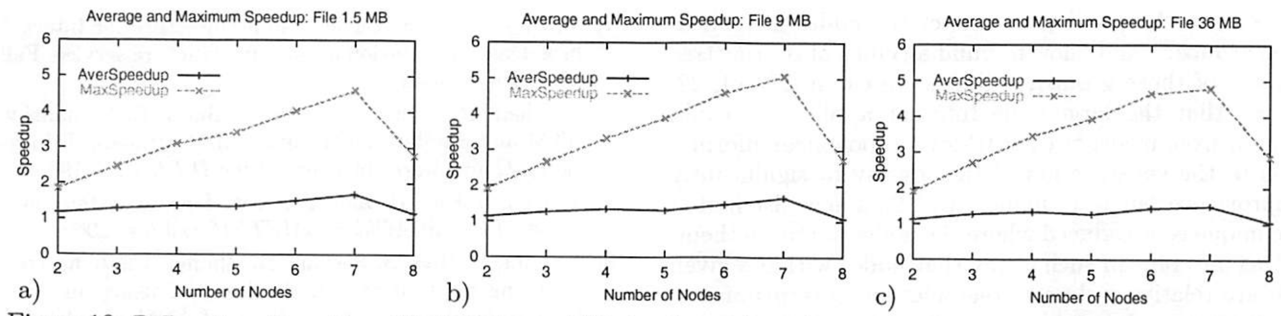


Figure 18: Different configuration with N_1 (*utexas.edu*) being the origin node: speedup in average and maximum replication times under *FastReplica* vs *Multiple Unicast* for a different number of nodes in replication set and a) 1.5 MB file, b) 9 MB file, c) 36 MB file.

experiments. For the 80 KB file, *FastReplica* is not efficient, and the replication time (both average and maximum) is higher than under *Sequential Unicast* and *Multiple Unicast*. For the 750 KB file, the replication time under *FastReplica* is better than under *Sequential Unicast* and *Multiple Unicast* strategies, and the average and maximum speedup shown in Figure 17 c) is again significant. These results help to outline the "border" parameters for new strategy usage: in our case study, *FastReplica* works most efficiently for replicating files larger than 0.5 MB. For $n > 8$, the "border" file size, where *FastReplica* works most efficiently, may increase correspondingly.

The results presented in Figure 16 show the significant speedup both in average and maximum replication times under the *FastReplica* strategy. The additional analysis reveals that the available bandwidth of the paths between the origin node N_0 (*hp.com*) and nodes $N_1, ..., N_7$ (universities' machines) is significantly lower than the cross bandwidth between nodes $N_1, ..., N_7$. only node N_8 has a limited incoming bandwidth from all the nodes $N_0, N_1, ..., N_7$, while the outgoing bandwidth from node N_8 to $N_1, ..., N_7$ is again significantly higher. In such a configuration, *FastReplica* utilizes the abundance of additional available bandwidth between the replication nodes in the most efficient way to produce the spectacular results.

It is interesting to see how *FastReplica* would perform when a different node with high bandwidth paths to the rest of the nodes is used as the origin node. We changed the configuration and made node N_1 (*utexas.edu*) to be the origin node, and rerun the experiments again.

Figure 18 shows the average and maximum speedup of file replication time under the proposed *FastReplica in the small* relative to the replication time of *Multiple Unicast* for files of 1.5 MB, 9 MB, and 36 MB, and a different number of nodes in the replication set in the new configuration.

In the new configuration, the average replication times under *FastReplica* and *Multiple Unicast* are similar, but the maximum replication time under *FastReplica* is still significantly better than the maximum replication time under *Multiple Unicast*.

The bandwidth analysis reveals that node *utexas.edu* is connected to the rest of the nodes via high bandwidth paths with low bandwidth variation across these paths. Our analysis in Section 3.3 with a specially designed example, where the bandwidth matrix BW is defined by equations (6), demonstrates that when the cross bandwidth between some replication nodes is significantly lower than the bandwidth of the original paths from N_0 to the recipient nodes $N_1, ..., N_8$ then *FastReplica* improves the maximum replication time but may have no significant improvement in average replication time.

5 Conclusion

In recent years, the Web and Internet services have moved from an architecture where data objects are located at a single origin server or site to the an architecture where objects are replicated across multiple, geographically distributed servers. Client requests for content are redirected to a best-suited replica rather than the origin server. For large files, the replication process across this distributed network of servers is a challenging and resource-intensive problem on its own.

In this work, we introduce *FastReplica* for efficient and reliable replication of large files in the Internet environment. *FastReplica* partitions an original file into a set of subfiles and uses a diversity of Internet paths among the receiving nodes to propagate the subfiles within the replication set in order to speedup the overall download time for the original content. We scale the algorithm by clustering the nodes in a set of replication groups, and by arranging efficient group communications among them, i.e. by building the overlay tree on top of those groups.

Through experiments on a prototype implementation, we demonstrate the efficiency of *FastReplica in the small* in a wide-area testbed. Since *FastReplica in the small* defines the iteration step in the general algorithm, these performance results set the basis for performance expectations of *FastReplica in the large*.

The interesting and important issues for future research are "how to better cluster the nodes in replication groups?" and "how to build an efficient overlay tree on top of those groups?" Recent research [19, 14, 22] shows that the large-scale Internet application could benefit from incorporating IP-level topological information in the construction of the overlay to significantly improve overlay performance. In [19], a new distributed technique is introduced where the nodes partition themselves into bins in such a way that nodes within a given bin are relatively close to one another in terms of network latency. It might be an interesting technique for clustering "close" nodes into replication groups in *FastReplica*.

To analyze and validate future optimization for *FastReplica*, a large-scale Internet environment or testbed is needed. In recent work [23], authors propose ModelNet as a comprehensive Internet emulation environment to evaluate Internet-scale distributed systems. A new initiative within the research community around PlanetLab [18] is aiming to build a global testbed for developing and accessing new network services. The introduction of such environments and large-scale testbeds will help to support interesting scalability experiments in the near future.

Acknowledgements: We would like to thank HPLabs summer interns who helped us to build an experimental wide-area testbed from their university machines: Yun Fu, Weidong Cui, Taehyun Kim, Kevin Fu, Zhiheng Wang, Shiva Chetan, Xiaoping Wei, and Jehan Wickramasuriya. Their help is highly appreciated.

Authors also would like to thank John Apostolopoulos for motivating discussions on multiple descriptions for streaming media and John Sontag for his active support of this work.

We would like to thank the anonymous referees for useful remarks and insightful questions, and our shepherd Srinivasan Seshan for constructive suggestions to improve the content and presentation of the paper.

References

[1] Allcast *http://www.allcast.com/*

[2] J. Apostolopoulos. Reliable video communication over lossy packet network using multiple state encoding and path diversity. *Proc. of Visual Communications and Image Processing (VCIP)*, January, 2001.

[3] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee. On multiple description streaming with content delivery networks. *Proc. of IEEE Infocom*, 2002.

[4] J. Byers, M. Luby, M. Mitzenmacher, A. Rege. A Digital Fountain approach to reliable distribution of bulk data. *Proc. of ACM SIGCOMM*, 1998.

[5] J. Byers, M. Luby, M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using Tornado codes to speedup downloads. *Proc. of IEEE Infocom*, 1999.

[6] J. Byers, J. Considine, M. Mitzenmacher, S. Rost. Informed content delivery across adaptive overlay networks. *Proc. of ACM SIGCOMM*, 2002.

[7] Y. Chawathe. Scattercast: an architecture for Internet broadcast distribution as an infrastructure service. Fall 2000, PhD thesis.

[8] L. Chen, B. Vickers, J. Kim, T. Suda, E. Lesnansky. ATM and Satellite Distribution of Multimedia Educational Courseware. In *Proc. of the IEEE ICC*, 1996.

[9] Y. Chu, S. Rao, H. Zhang. A case for end system multicast. *Proc. of ACM SIGMETRICS*, June, 2000.

[10] Y. Chu, S. Rao, S. Seshan, H. Zhang. Enabling conferencing applications on the Internet using an overlay multicast architecture. *Proc. of ACM SIGCOMM*, 2001.

[11] H. Deshpande, M. Bawa, H. Garcia-Molina. Streaming live media over peer-to-peer network. Technical Report, Stanford University, August, 2001.

[12] P. Francis. YOID: Your Own Internet Distribution. http://www.aciri.org/yoid/. April, 2000.

[13] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, J. O'Toole, Jr. Overcast: Reliable Multicasting with an Overlay Network. *Proc. of the 4th Symp. on Operating System Design and Implementation (OSDI)*, 2000.

[14] D. Kostic, A. Rodriguez, A. Vahdat. The Best of Both Worlds: Adaptivity in Two-Metric Overlay Networks. Duke University Technical Report, May, 2002.

[15] T. Nguyen and A. Zakhor. Distributed Video Streaming over the Internet. *Proc. of SPIE Conference on Multimedia Computing and Networking*, San Jose, CA, 2002.

[16] V. Padmanabhan, H. Wang, P. Chou, K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. *Proc. of 12th ACM NOSSDAV*, May, 2002.

[17] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: an application level multicast infrastructure. *Proc. of the 3d USENIX Symposium on Internet Technologies*, March, 2001.

[18] PlanetLab. http://www.planet-lab.org/

[19] S. Ratnasamy, M. Handley, R. Karp, S. Shenker. Topologically-aware construction and server selection. *Proc. of IEEE Infocom*, 2002.

[20] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the Internet. *Proc. of IEEE Infocom*, 2000.

[21] P. Rodriguez, E. W. Biersack. Bringing the Web to the Network Edge: Large Caches and Satellite Distribution. ACM Special Issue. In *Journal on Special Topics in Mobile Networking and Applications (MONET)*, 2001 .

[22] A. Roy, S. Das. Optimizing QoS-Based Multicast Routing in Wireless Networks: A Multi-Objective Genetic Algorithmic Approach. *Proc. of Second International IFIP-TC6 Networking Conference (Networking'2002)*, LNCS vol. 2345, Springer-Verlag, Pisa, Italy, 2002.

[23] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *Proc. of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[24] vTrails. *http://www.vtrails.com/*

Energy Conservation Policies for Web Servers [*]

Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony
Low-Power Computing Research Center
IBM Research, Austin TX 78758, USA.
http://www.research.ibm.com/arl

Abstract

Energy management for servers is now necessary for technical, financial, and environmental reasons. This paper describes three policies designed to reduce energy consumption in web servers. The policies employ two power management mechanisms: dynamic voltage scaling (DVS), an existing mechanism, and request batching, a new mechanism introduced in this paper. The first policy uses DVS in isolation, except that we extend recently introduced task-based DVS policies for use in server environments with many concurrent tasks. The second policy uses request batching to conserve energy during periods of low workload intensity. The third policy uses both DVS and request batching mechanisms to reduce processor energy usage over a wide range of workload intensities. All the policies trade off system responsiveness to save energy. However, the policies employ the mechanisms in a feedback-driven control framework in order to conserve energy while maintaining a specified quality of service level, as defined by a percentile-level response time.

We evaluate the policies using Salsa, a web server simulator that has been extensively validated for both energy and response time against measurements from a commodity web server. Three day-long static web workloads from real web server systems are used to quantify the energy savings: the Nagano Olympics98 web server, a financial services company web site, and a disk intensive web workload. Our results show that when required to maintain a 90th-percentile response time of 50ms, the DVS and request batching policies save from 8.7% to 38% and from 3.1% to 27% respectively of the CPU energy used by the base system. The two policies provide these savings for complementary workload intensities. The combined policy is effective for all three workloads across a broad range of intensities, saving from 17% to 42% of the CPU energy.

*This research has been supported in part by The Defense Advanced Research Projects Agency under contract F33615-00-C-1736.

1 Introduction

There are many technical, financial, and environmental motivations to reduce server energy consumption in Internet data centers. In this paper, we describe three policies for reducing energy consumption of server processors during web serving, a common application in data centers. The policies employ two energy management mechanisms: dynamic voltage scaling, an existing mechanism, and *request batching*, a new mechanism we introduce in this paper. All three policies save energy while maintaining server responsiveness at or better than a specified quality of service level. We evaluate the policies using Salsa, a web server simulator that has been extensively validated for accuracy in both energy consumption and response times against measurements from a commodity server. The evaluation uses three day-long static web serving workloads.

The first policy uses dynamic voltage scaling (DVS), a mechanism that lets the processor frequency and voltage be varied dynamically. DVS-based policies leverage the fact that a given task can be completed for less energy if executed at a lower processor frequency and voltage. We extend recently introduced task-based DVS policies [7, 14] for use in environments with many concurrent tasks, such as in web servers. The second policy is based on a new mechanism called *request batching* that groups requests received by the server and executes them in batches, placing the server processor in a low-energy state between batches. These policies are complementary in two respects: they require different kinds of support from the system hardware and are effective over different ranges of workload intensities. The third policy combines both request batching and dynamic voltage scaling mechanisms to conserve energy across a broader range of workload intensities than the individual policies. All three policies trade off system responsiveness in order to save energy. However, the policies employ a feedback-driven control framework in order to conserve energy while maintaining a given quality of service level, as defined by a percentile–level response time.

The request batching and dynamic voltage scaling policies target complementary ranges of workload intensities: for a given QoS level, request batching works well when the workload is light, while DVS yields more savings as the workload intensifies. The combined policy reduces energy consumption over a broad range of workload intensities. For the range of workloads and intensities examined in this paper, we find that request batching provides from 3.1% to 27% savings in CPU energy consumption while dynamic voltage scaling provides from 8.7% to 38% savings. The combined policy provides savings in CPU energy consumption ranging from 16.6% to 41.9%. All these savings are realized while maintaining the 90th–percentile first–packet response times at or better than 50ms.

This paper makes the following contributions:

- Presents a DVS–based policy for use in server environments with concurrent tasks, an evolution of existing task–based DVS policies [7, 14].
- Introduces a new energy management mechanism, request batching, and a policy that employs it.
- Leverages the DVS and request batching mechanisms into a combined policy that achieves energy savings across a broad range of workload intensities.
- Demonstrates that by using a feedback–driven framework, the policies achieve significant energy savings while maintaining system responsiveness at or better than a desired level.

The rest of this paper is organized as follows. Section 2 explains why energy management is crucial for Internet data centers. Section 3 contains a description of the three policies. Section 4 describes the workloads used in this study, and Salsa, the simulator we use for evaluating the policies. The three policies are evaluated in Section 5. A comparison to related work is presented in Section 6. Finally, Section 7 concludes the paper.

2 Why Manage Energy Usage in Data Centers?

There are many technical, financial, and environmental motivations to reduce server energy consumption. For instance, data centers deploy thousands of servers, densely packed to maximize floor space utilization. Such dense deployment pushes the limits of power delivery and cooling systems. Anecdotal evidence from data center operators points to the intermittent failures of computing nodes in densely packed systems due to insufficient cooling. Furthermore, constraints on the amount of power that can be delivered to server racks makes energy conservation critical for fully utilizing the available space on these racks.

Beyond these technical motivations for reducing energy usage, there are two compelling financial incentives for data centers in the United States to manage server energy usage. Similar financial arguments may also apply in other parts of the world. First, data centers typically bundle energy costs into the cost they charge consumers for hosting. For instance, the average price to rent a full rack of space (about 3ft × 3ft × 6ft) at a data center is approximately US$1000 per month as of September 2002[1]. For this price, the data center provides physical space, energy, energy backup (UPS), cooling, and support services (offices). Bandwidth is metered and is usually an extra charge. While the amount of power provided per rack varies from one data center to another, all centers include at least 4KW of power per rack, with some delivering up to 7KW. Thus, with energy costs averaging 8 cents per kWh, rack energy usage alone could account for up to 23% to 50% of colocation *revenue*. Similar arguments apply for managed hosting (where the data center controls and owns the IT equipment and provides the service) as well. Consequently, there is a strong incentive to minimize server energy usage in data centers.

Second, faced with the prospect of constructing new facilities in order to meet the electricity demands of data centers, many utilities have instituted rate tariffs or upfront deposits [16, page 56]. For example, Puget Sound Energy's schedule 449 requires the customer to bear costs associated with dedicated generation and/or delivery facilities [6]. Since a data center can always supplement utility-provided power with on-site generation, a center could lower its capital outlay by placing lower demands on the utility and employing on-site generation during periods of

[1]Approximate average colocation price charged by hosting centers such as Mzima (www.mzimahosting.com/prodserv/promotion1.html), XMission (www.xmission.com/business/colocation.html), and Terracom Network Services (www.tns.net/internet/colo.html?gl).

peak power demand. Reducing the average energy usage is critical for this strategy to be successful.

3 Energy Management Policies

Our policies focus on reducing server CPU energy consumption. While CPU energy is only one component of the system energy, our previous research, based on measurements from an actual commodity system that typifies servers currently deployed in Internet Data Centers, determined that the CPU is the dominant consumer of system energy [2]. Furthermore, when considering active system components, the CPU exhibits the most variation in energy consumption. Finally, these policies are applicable to individual web server systems and complement energy management techniques for server clusters [4, 5, 19].

3.1 Feedback–driven Control Framework

All three energy management policies use a feedback–driven control framework to maintain the system responsiveness at a specified level. The system administrator establishes a percentile–based response time goal. A 90th–percentile goal means that 90% of the requests received over a specific interval must have a response time equal to or better than the specified goal. In our policies we compute the 90th–percentile response time over an entire day, but other intervals could be used (see Section 5.1). The server continuously monitors the response time of individual requests as measured by the difference between the time the request was received at the server and the time the first packet of the response was sent to the client. The feedback–driven control framework adjusts policy parameters to increase energy savings when measured response times are lower (better) than the response time goal, or to decrease energy savings when system is not meeting its response time goal. This is done on a best-effort basis and the system may not be able to meet the goal under conditions of extreme load.

When evaluating the policies in this paper, we mostly use a 90th–percentile response time goal of 50ms. Relaxing the server response time to 50ms will have minimal effect on the client perceived response times (CPRT). This is because the CPRT is typically dominated by wide area network overheads [20], enabling a 50ms server response time to be masked by the WAN delays. As a result, the web server appears responsive to the end–user even if it takes up to 50ms to respond to requests. We also show how the energy savings change as both the percentile and response time goal are varied.

While other responsiveness metrics (such as the average response time) could be used, most service level agreements are crafted based on a percentile goal (for instance, see the BS Web Services SLA [22]). The response times for individual requests can be ascertained through a combination of the OS tagging incoming packets with their arrival time, and the web server notifying the OS when requests are serviced. Current versions of Linux already tag incoming packets with their arrival time and support the SIOCGSTAMP ioctl which returns the arrival time of the last packet passed to the application on a specified socket. Web servers commonly generate a time stamp for each request completion, since this information is typically recorded in the web server access log. Thus, computing server response time should require at most one additional ioctl and a simple calculation for each request. Note that the control mechanism only needs to determine the percentage of responses that meet the target, as opposed to actually computing the 90th–percentile response time, which can be computationally expensive.

3.2 Dynamic Voltage Scaling Policy

Dynamic voltage scaling (DVS) policies reduce energy consumption by varying the processor operating point (frequency and voltage) according to the rate at which work must be done. Recent research in DVS policies set the processor operating point using a task–based approach. For instance, Flautner et. al. employ a policy that sets CPU speed on a per–task basis [7]. Lorch and Smith describe an algorithm for improving the performance of task-based DVS policies when task completion times cannot be accurately predicted [14]. These techniques perform well for desktop application workloads but are unsuitable for environments with many concurrent tasks, such as server systems. Consequently, while our DVS policy keeps track of the response time of individual requests ("tasks"), it employs a feedback–driven control framework (explained in Section 3.1) to meet responsiveness requirements in the aggregate, instead of for individual tasks. The

DVS policy adjusts the processor operating point at regular intervals or *quanta* to meet the overall responsiveness requirement.

At the beginning of each quantum, the DVS policy selects an operating point (voltage and frequency) for the next quantum based on the response times for all previously serviced requests. If the system is more responsive than required, the processor frequency is decreased by one step (if not already at its minimum) and the voltage is set accordingly. If the response time goal is not being met, the frequency is increased by one step (if not already at the maximum) after the core voltage is sufficiently raised.

Dynamic voltage scaling provides the most energy benefits for moderately intense workloads. Since the processor operating point cannot be lowered below a certain level, DVS provides few benefits for low intensity workloads. Likewise, DVS yields little benefit for very heavy workloads, since the processor must run mostly at full speed to meet the responsiveness requirement. The operating point that yields the most savings depends on the processor parameters and the DVS policy. For instance, with the DVS processor we consider in this paper (see Section 4.2 for the parameters), a DVS policy that keeps the processor fully utilized by adapting the operating point to the available load achieves its maximum energy savings for a load that is approximately 58% of the processor's capacity at its highest frequency setting.

3.3 Request Batching Policy

Policies based on dynamic voltage scaling are not very effective at low workload intensities. However, previous studies have observed that Web servers are relatively idle for large fractions of time [13]. Even when idle, server processors consume significant amounts of power. Unfortunately, since incoming requests arrive asynchronously, web servers cannot afford to use energy conserving states with significant wakeup penalties such as the "hibernation" mode commonly found in laptops.

Request batching is a mechanism that we have developed to conserve energy during periods of low workload intensities. In request batching, the servicing of incoming packets from the network is delayed while the main processor of the web server is kept in a low power state. Incoming packets are accumulated in memory until a packet has been kept pending for longer than a specified *batching timeout*[2]. Request batching saves energy because while requests are being accumulated, the processor can be placed in a lower power state such as Deep Sleep [11, Page 82] instead of just idling it. Furthermore, since web servers are typically idle or at low utilization much of the time, the energy saved during idle periods can result in significant energy savings. For example, if placing the processor in Deep Sleep mode reduces power consumption by 2.5 Watts, a web server that is on average only 25% utilized could save 162 KJ of energy per day.

Request batching provides the most energy saving benefits for light workloads. For a given batching timeout, the savings from request batching decrease with increasing workload intensity (until the processor becomes fully utilized) [3]. Increasing the batching timeout saves more energy at the expense of increased response time. In order to meet the specified quality of service (system responsiveness) level, we use a feedback–driven control framework (explained in Section 3.1) to dynamically adapt the batching timeout.

When a processor is in Deep Sleep, it requires the delivery of a specific set of signals in order to be reactivated. Network adapters that support the Wake-on-LAN feature [10] are already capable of waking up a processor in Deep Sleep, and the same mechanisms could be employed here. Furthermore, most network adapters are capable of DMA-ing incoming packets into pre-specified buffers in memory. Similarly, the disk can process pending commands and DMA data into memory. The experience we gained with our prototype (described in Section 4.2.1) indicates that batching timeouts of up to 100ms will not adversely affect TCP performance.

The request batching policy places the processor into Deep Sleep mode when there are no pending requests to be serviced. The policy adjusts the length of the batching timeout based on the system responsiveness. The batching timeout is varied in steps of

[2]We also considered using a *maximum request backlog*, but discovered that the timeout, by itself, provides the level of control needed.

[3]Consider requests arriving at λ/sec, each taking τ seconds to process. Ignoring the time taken to wake up from Deep Sleep, the energy savings from batching will be $(1 - \lambda\tau)(P_{idle} - P_{DS})$ per unit time. Expressed as a fraction over the base energy, the batching savings are: $(1 - \lambda\tau)(P_{idle} - P_{DS})/[P_{max}\lambda\tau + P_{idle}(1 - \lambda\tau)]$. This decreases sublinearly as the request rate increases.

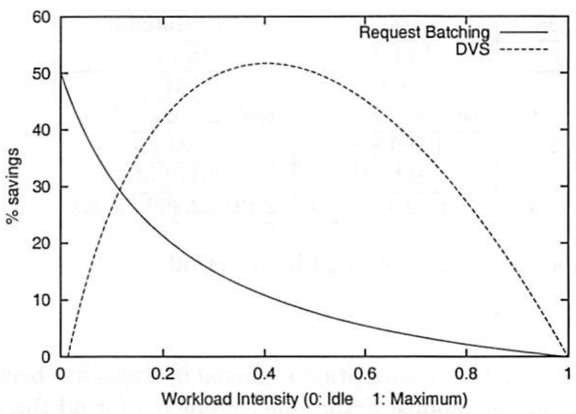


Figure 1: Energy savings (over the base case) from DVS and request batching for a range of workload intensities.

10ms. If the system is more responsive than needed, the timeout is increased one step. If it is not meeting the specified responsiveness target, the timeout is decreased one step. Since the processor operates at full power during the wakeup sequence (≈1.5ms), it is not advantageous to decrease the timeout below a certain limit. If the timeout falls below this limit, the request batching mechanism is disabled until the timeout is again larger than this limit.

3.4 Combined Policy

Dynamic voltage scaling and request batching target different workload intensity ranges on the server. Figure 1 illustrates this behavior graphically. To handle a range of workload intensities beyond what each policy can handle individually, we have devised a combined policy that leverages both request batching and dynamic voltage scaling mechanisms. The combined policy invokes the request batching and voltage scaling mechanisms based on the activity level of the server and reduces energy consumption across a broad range of workload intensities while maintaining system responsiveness at the required level.

When there are no pending requests to be serviced, the processor is placed into DeepSleep mode as in request batching. This policy adjusts the batching timeout as in the request batching policy. When the processor wakes up from Deep Sleep, it is placed at the lowest operating frequency. From this point onwards, the policy adjusts the processor frequency and voltage as in the DVS policy.

4 Methodology

4.1 Workloads

We constructed workloads using web server logs obtained from several production Internet servers. The first workload, Olympics98, is derived from the requests received on Feb 19, 1998 at one geographically replicated facility hosting the 1998 Winter Olympics web site. The second workload, Finance, is derived from the requests recorded on Oct 19, 1999 at the web site of a major financial services company. The third workload, referred to as Disk-Intense, is derived from the May 2, 2001 log of the Silicon Valley (SV) proxy server operated by the Information Resource Caching (IRCache) Project [12]. Strictly speaking, a proxy server is not a web server in that it does not have its own content, but we included this workload to represent a scenario of a web server with extremely low cache locality and thus high levels of disk activity. A complete description of the methodology for generating workloads from server access logs is given elsewhere [2].

The characteristics of the three workloads are summarized in Table 1. The "peak requests per second" is the highest observed rate for a one minute period. The "average requests per connection" is based on our technique of grouping requests into connections [2]. The amount of memory needed to hold the unique data for 97%/98%/99% of all requests is tagged by that moniker.

From the three base workloads, we generate workloads representing a range of client load intensities by scaling the inter-arrival time of connections by a constant amount. A scalefactor of "2.5×" corresponds to reducing the inter-arrival time of connections by a factor of 2.5. While this scaling increases the connection arrival rate by the scalefactor, it maintains the same basic pattern of connection arrivals. We preserve the inter-arrival times of requests within a connection since these represent user think times or network/client overheads in retrieving web page components. Chase et. al. have adopted a similar approach [4].

4.2 Salsa: A Validated Web Server Simulator

In order to evaluate the policies, we have constructed Salsa, a simulator that estimates the en-

Workload	Olympics98	Finance	Disk-Intense
Avg requests (Peak requests) / sec	97 (171)	16 (46)	15 (30)
Avg requests / connection	12	8.5	31
Unique files (Total file size)	61,807 (705MB)	16,872 (171MB)	698,232 (6,205MB)
Distinct HTTP Requests	8,370,093	1,360,886	1,290,196
Total response size (excl HTTP headers)	49,871 MB	2,811 MB	10,172 MB
97%/98%/99% (MB)	24.8 / 50.9 / 141	3.74 / 6.46 / 13.9	2,498 / 2,860 / 3,382

Table 1: Characteristics of three web server workloads over a 24-hour period.

ergy consumption and response time of a web server. Salsa is based on a queuing model built using the CSIM execution engine [15] and determines the CPU time and energy taken to service web requests using a model that is parameterized with energy measurements from a commodity, 600MHz Intel processor–based web server. Among other events, Salsa models process scheduling and file cache hits and misses. At present, Salsa does not model disk delays, but models the extra energy expended by the CPU (due to driver code execution) while making disk accesses.

The processor we simulate with Salsa has maximum power consumption P_{max} = 27.2 Watts, idle power consumption P_{idle} = 4.97 Watts, and Deep Sleep power consumption $P_{DeepSleep}$ = 2.47 Watts. To evaluate the DVS policy, we scaled data obtained from a low–power processor to fit the 600MHz Intel processor's maximum frequency and core operating voltage [18]. The DVS processor has a frequency range of 300MHz - 600MHz variable in steps of 33MHz, a time quantum of 20ms for varying frequency and voltage, and a core operating voltage ranging from 1.5V to 2V. The P_{max} and P_{idle} of the DVS processor are the same as that of the 600MHz non–DVS Intel processor.

Our decision to use a simulator to evaluate the policies was motivated entirely by pragmatic concerns. If we had replayed the traces against a real web server that implemented the policies, it would have taken us over 55 days to create all the data points in this paper [4]. Faster computers could not have speeded up this time. In contrast, the Salsa simulations on a 1GHz Intel machine took under 4 hours of wall clock time. However, as we describe below, we have validated Salsa against a real web server to ensure that its results can be meaningfully interpreted.

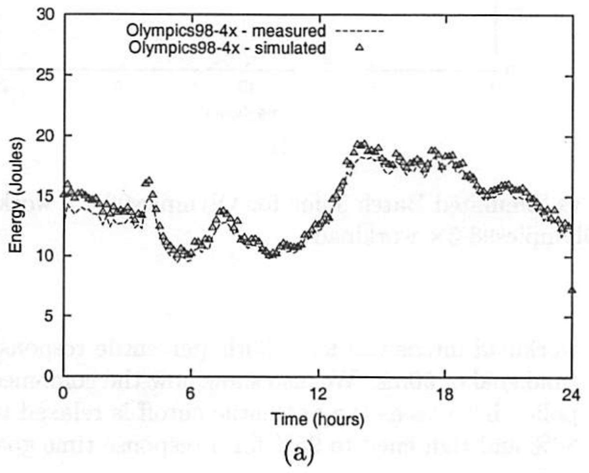
[4]Figure 4(a) alone would have taken 2×(10 points × 12 hrs + 10 points × 2 hrs + 9 points × 6 hrs) = 388 hours (over 16 days).

The energy consumption reported by Salsa has been validated against actual measurements for all three workloads (none of these workloads were used to calibrate Salsa). We used a modified version of `httperf` [17] to replay the workload against the server. Figure 2(a) shows the measured CPU energy consumed by the 600MHz system during the execution, overlaid with the simulator output. The Finance and Disk-Intense workloads exhibit similar behavior. Table 2 summarizes the validation results over a broad range of intensities, from light (Disk-Intense-2×, Finance-12×) to heavy (Olympics98-4×). For all three workloads, the error in predicted energy is less than 6%.

We have also validated the response times predicted by Salsa for the Olympics98 and Finance workloads against those measured using the `httperf` tool. Figure 2(b) shows the response time predicted by the simulator overlaid with the measured response times for the Olympics98 workload at a 5× intensity (we show data for the 5× intensity because the response time shows little variation at lower intensities). For both workloads, the average response time predicted by the simulator has an error of at most 13.2% compared to response times measured during execution. This error arises because the measured response time data involves three components: the server, the network, and the client, of which Salsa is modeling only the server. Since Salsa does not yet model disk delays, we have not attempted to validate the response times predicted by Salsa for the disk–intensive workload. However, when response time is dominated by disk access time, any increase in response time due to CPU power management will be marginal.

Workload	Olympics98-4×	Finance-12×	Disk-Intense-2×
Measured CPU Energy (Joules)	1,232,710	711,415	627,977
Simulator CPU Energy (Joules)	1,253,652	739,200	663,648
Error in Total Energy	1.70%	3.91%	5.68%
Correlation Coefficient	0.9846	0.9960	0.8485

Table 2: Comparison of Measured to Simulated CPU energy for three workloads. Correlation coefficients were computed based on the energy used in 30 second intervals over the length of the run.



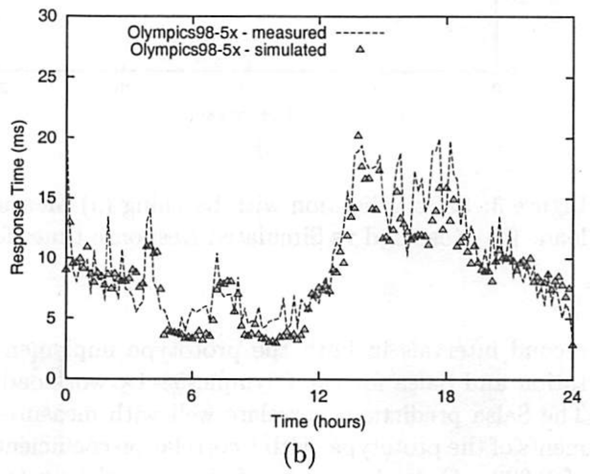


Figure 2: (a) Measured vs Simulated Energy Consumption for Olympics98-4× workload. (b) Measured vs Simulated Response times for Olympics98-5× workload.

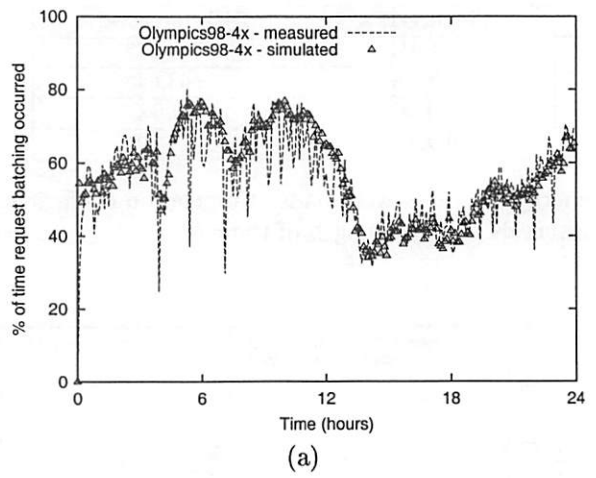
4.2.1 Salsa Validation For Request Batching

Before simulating the request batching policy in Salsa, we wanted to gain a measure of confidence in the simulation technique we used. We achieve this confidence by validating Salsa's predictions against a prototype implementation of request batching.

The request batching prototype is implemented on a commodity 600MHz server running the Apache web server over a Linux 2.4.3 kernel. We modified the web server to batch requests whenever all the Apache child processes were in an idle state: either waiting for a new connection or waiting for new requests on an existing connection. Batching is disabled when new work arrives until all child processes become idle again. We batch packets by leveraging the interrupt coalescing feature in the Alteon ACEnic Gigabit Ethernet card [1] on the web server. Interrupt coalescing groups the interrupts for a set of packet events (reception or transmission) into one single interrupt to the host processor and was introduced to improve network throughput by reducing interrupt processing overheads [1]. We modified the ACEnic device driver to allow the packet count and timeout for interrupt coalescence to be modified dynamically using an `ioctl` call.

To validate Salsa, we conducted an experiment where the prototype batches requests with a timeout of 100 milliseconds or a maximum backlog of 100 packets. There are two key differences between the prototype and the Salsa implementation we discuss in the later sections. First, the prototype does not place the processor in Deep Sleep. Instead, we keep track of the time that the processor *would have been* in Deep Sleep mode, since this is a direct indicator of the energy savings from batching. Second, the prototype does not incorporate the response time based feedback control explained in Section 3.1. Consequently, for validation purposes, we execute Salsa in "open–loop" mode without the feedback control, and compare the predicted batching and response times against the measured values.

Figure 3(a) shows the percentage of time for which request batching was enabled averaged over 90–



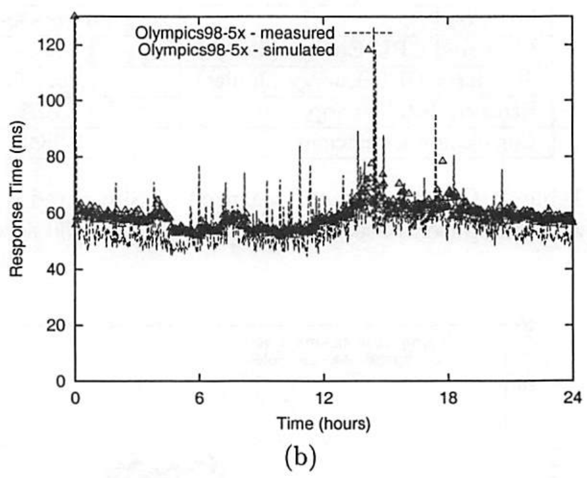


Figure 3: Salsa validation with batching (a) Measured vs Simulated Batch Time for Olympics98-4× workload. (b) Measured vs Simulated Response times for Olympics98-5× workload.

second intervals in both the prototype implementation and Salsa for the Olympics98-4× workload. The Salsa predictions correlate well with measurements of the prototype, with a correlation coefficient of 0.828. Over the course of the run, the prototype batched requests for a total of 11,953 seconds while Salsa predicted 12,373 seconds, an error of just over 3.5%. Figure 3(b) shows the client response time, averaged over 30–second intervals, for both the prototype and Salsa executing the Olympics98-5× workload. The correlation coefficient in response times is 0.754. The average response time predicted by the simulator has an error of 4.7% compared to that measured using the prototype. The same difficulties for validating response times that were outlined in Section 4.2 apply here as well. Finally, even in a LAN environment, we observed that the prototype experienced extra TCP retransmissions of less than 0.1% for a batching timeout of 100ms.

5 Evaluation

We begin by evaluating the DVS and request batching policies, explaining first how much energy each policy is able to save when the 90th–percentile response time goal is varied. Next, we set the 90th–percentile response time goal to be 50ms and evaluate how both policies behave in terms of energy savings as the workload intensity is changed. Finally, we evaluate the combined policy showing how it is able to sustain energy savings across different workload intensities for a 90th-percentile response time goal of 50ms. We also show how the combined policy behaves as the percentile cutoff is relaxed to 80% and tightened to 95% for a response time goal of 50ms.

5.1 Dynamic Voltage Scaling Policy

Figure 4(a) shows the energy savings from the DVS policy over the base system for a range of 90th–percentile response time goals. The three workloads exhibit different energy savings because of differing workload intensities. Relaxing the response time goal increases the amount of energy savings, up to a point. For example, the energy savings for the Disk-Intense-2× and Finance-12× workloads level off at response time goals higher than 20ms. In fact, the 20ms response time goals for these two workloads can be satisfied by running the DVS processor at no more than the minimum frequency of 300MHz. In contrast, for a heavier workload such as Olympics98-4×, the energy savings diminish only after the 90th–percentile response time goal is relaxed to about 100ms. This heavier workload benefits from DVS to a greater extent. Table 3 shows the energy consumption with and without DVS for the three workloads when the 90th–percentile response time goal is 50ms.

Fortunately, the figure shows that the extent to which the response time goal must be relaxed to capture the major fraction of the energy savings is well within acceptable QoS norms. Relaxing the

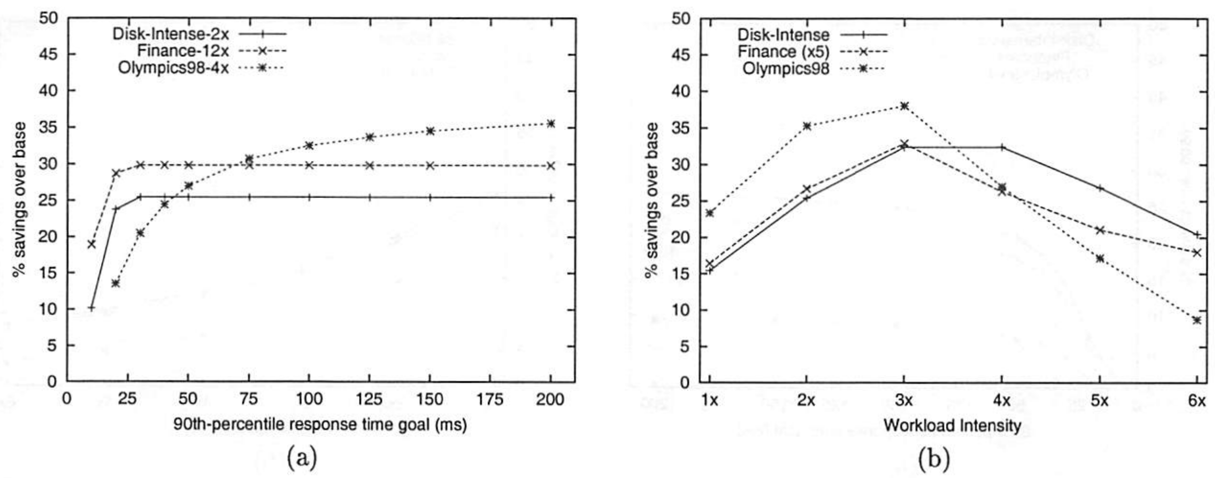


Figure 4: (a) Dependence of DVS energy savings on the response time goal. Note: Olympics98-4× does not have a 10ms data point, since the baseline 90th-percentile response time is itself 12.3ms. (b) Dependence of DVS energy savings on workload intensity for a 50ms 90th-percentile response time goal.

goal beyond the knee of the curve provides little improvement in energy savings at the expense of worsened responsiveness.

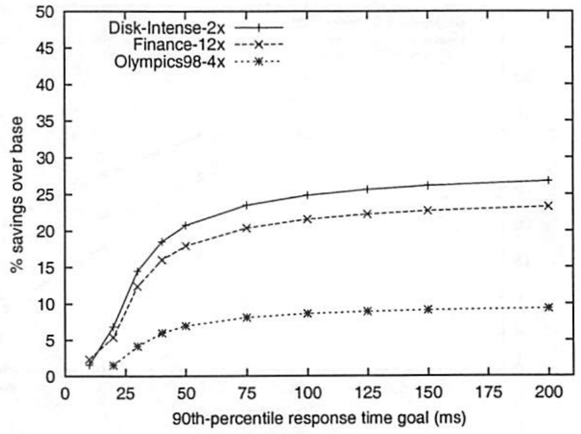
Figure 4(b) depicts the effect of workload intensity on the energy savings obtained from the DVS policy for a 90th-percentile response time goal of 50ms. Here, the x-axis indicates the scalefactor of the Olympics98 and Disk-Intense workloads, and the scalefactor divided by 5 for the Finance workload. Consistent with the analysis above, the results show that the energy savings from DVS improve with increasing workload intensity – but only up to a point. The largest savings are obtained when the intensity is between the light and heavy extremes. Changing the response time goal affects the magnitude of the savings, but not the intensity at which the gains are maximized.

Despite measuring 90th-percentile response time over an entire day, our control framework is surprisingly robust. Even for the most intense workloads we consider here, the 90th-percentile goal was attained for 84%, 86%, and 89% respectively of all 5-minute intervals of the Olympics98, Finance, and Disk-Intense workloads. A more complete discussion of the effect of the control system parameters on the system responsiveness is beyond the scope of this paper.

The results make two important points. First, for moderately intense workloads, dynamic voltage scaling can provide significant energy savings. Second, dynamic voltage scaling is less effective for workloads that are either light or intense.

5.2 Request Batching Policy

Figure 5(a) illustrates the energy savings due to request batching as the 90th-percentile response time goal is varied from 10ms to 200ms. The response time goal has a significant effect on the energy savings from request batching. Consistent with the analysis presented in Section 3.3, our simulation results show that increasing the response time goal yields diminishing returns in energy savings. Table 3 shows the energy consumption with and without request batching for a 90th-percentile response time goal of 50ms. The Finance and Disk-Intense workloads exhibit higher savings because of the prolonged periods of very low request rates in these workloads. For example, the Finance workload has a very low request rate until about 9:30am (see Figure 6). Request batching achieves high energy savings by placing the processor in the Deep Sleep state for a significant fraction of this time. In contrast, the Olympics98 workload has relatively fewer periods where requests can be batched. Compared to the 90th-percentile response time without request batching, the 50ms goal is several factors larger. However, as explained in Section 3.1, the effect on the client-perceived response time will be minimal due to the CPRT's dependence on the WAN delay between the client and the server.



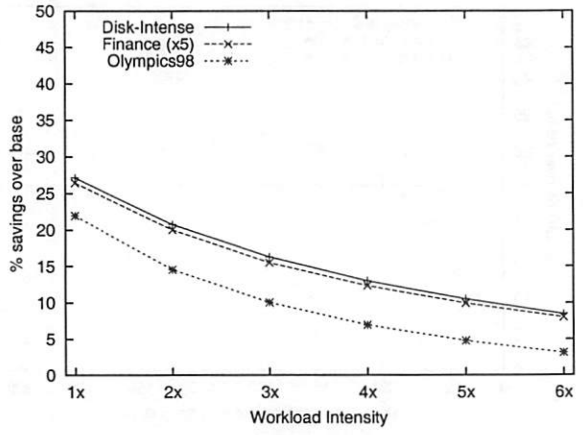


Figure 5: (a) Dependence of request batching energy savings on the response time goal (b) Dependence of request batching energy savings on workload intensity for a 50ms 90th–percentile response time goal.

Workload	Olympics98-4×	Finance-12×	Disk-Intense-2×
Base energy – no DVS or Batching (J)	1,253,672	739,212	663,648
Base 90th–percentile response time (ms)	12.3	6.4	3.0
DVS Joules (% savings)	915,204 (27%)	518,844 (30%)	494,982 (25%)
Request batching Joules (% savings)	1,166,128 (7.0%)	606,468 (18.0%)	525,836 (20.8%)

Table 3: Energy savings from DVS and Request Batching for a 90th–percentile response time goal of 50ms.

Figure 5(b) illustrates the energy savings with a 90th–percentile response time goal of 50ms compared to the base case where requests are not batched. The x-axis represents the workload intensity for the Olympics98 and Disk-Intense workloads, and the intensity divided by 5 for the Finance workload. Note that the savings decrease with intensity, as explained in Section 3.3. As in the case of DVS, changing the response time goal affects the resultant energy savings, but does not alter where the gains from request batching are maximized.

In summary, request batching is an extremely effective strategy for saving energy when the workload has significant periods of low activity. While increasing the response time goal has a positive effect on the energy savings, most of the savings can be captured using a response time goal of around 50ms, without adversely affecting either the client-perceived response time or TCP/IP performance.

5.3 Combined Policy

Figure 6 shows the savings from DVS and request batching superimposed over the request rate during the course of the Finance-12× workload for a 90th–percentile response time goal of 50ms The figure clearly shows the effectiveness of request batching at low request rates or workload intensities, e.g., until 9:30am. After a steep climb at 9:30am, the request rate remains relatively high until about 6pm. During this interval, request batching provides reduced energy savings. In contrast, it is precisely during this period that DVS provides maximum savings. This complementary behavior clearly illustrates the potential benefits of a policy that employs both mechanisms for improved energy savings across a broad range of workload intensities.

Figure 7 compares the energy savings from the DVS, request-batching, and combined policies for the Disk-Intense workload at intensities from 1× to 6×. We can see the DVS and request-batching mechanisms complementing each other: when the workload is light, request batching provides the most savings. As the workload becomes more and

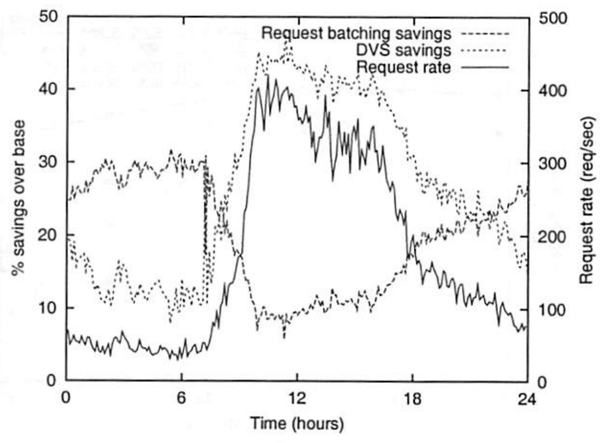


Figure 6: Energy savings from DVS and request batching with a 90th–percentile response time goal of 50ms for Finance-12×, superposed with the request rate (intensity). All values averaged over 6–minute windows.

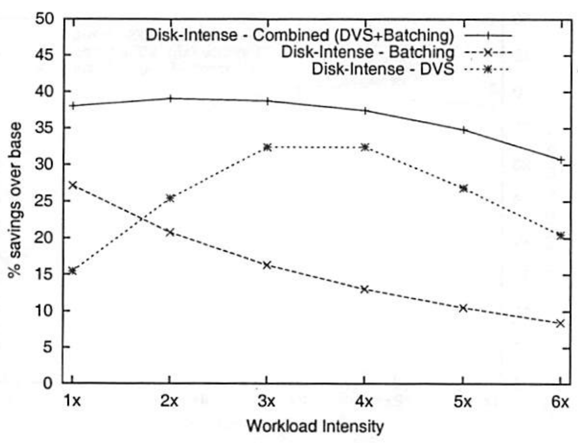


Figure 7: Energy savings from the DVS, request batching, and combined policies with a 90th–percentile response time goal of 50ms for Disk-Intense over a range of intensities.

more intense, requests are batched less and less frequently (if at all), and DVS provides the most savings. The combined policy provides savings of between 30.7% and 39% as the workload intensity varies from 1× to 6×, illustrating that the combination of request batching and voltage scaling captures the best features of both policies.

Figure 8(a) shows the energy savings yielded by the combined policy on all three workloads. The combined policy outperforms the individual request batching and voltage scaling policies across the range of workloads and intensities. However, while the energy savings for both Finance and Olympics98 are above 40% at low workload intensities, the savings drop to about 26% and 17% respectively as the workload intensifies to a scale of 6. The savings decrease because higher intensities require more processor involvement in order to service incoming requests. In particular, the baseline 90th–percentile response time (with no energy saving policy in effect) for the Olympics98-6× workload is 42ms, implying that the system has little room to save energy while maintaining the 90th–percentile response time at 50ms.

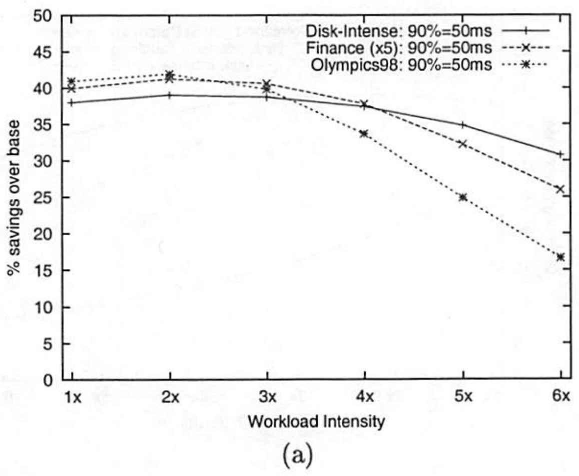
Figure 8(b) shows the variation in savings as the percentile goal is varied from 80% to 95% for the Olympics98 and Disk-Intense workloads. The Finance workload exhibits similar behavior. As the percent of response times that must fall at or below 50ms is tightened from 80% to 95%, the energy savings decrease. In the case of Disk-Intense at 6× intensity, the savings fall from 33.5% to 9.5%. This experiment illustrates that energy savings can be obtained not only by relaxing the response time goal, but also by relaxing the percent of requests that must be satisfied within that goal.

5.4 Projections for Faster Processors

The results we report in this paper are for a simulated server with a 600MHz processor. Current processors use clock rates as high as 3.0 GHz, and even faster processors will be available in the near future. In this section we project how our three energy management policies will perform in systems with faster processors. We use a hypothetical 3.0 GHz processor model that supports a low–power Deep Sleep mode and dynamic voltage scaling, and perform simulations using our three web server workloads. The key attributes of our 3.0 GHz processor

Processor State	Power Consumption
Busy (at 3GHz)	60W
Idle (Halted)	10W
Deep Sleep	5W
DVS Frequency Range	
1.5 GHz - 3.0 GHz, in 10 steps of 150MHz	

Table 4: Attributes of a hypothetical 3.0 GHz DVS processor



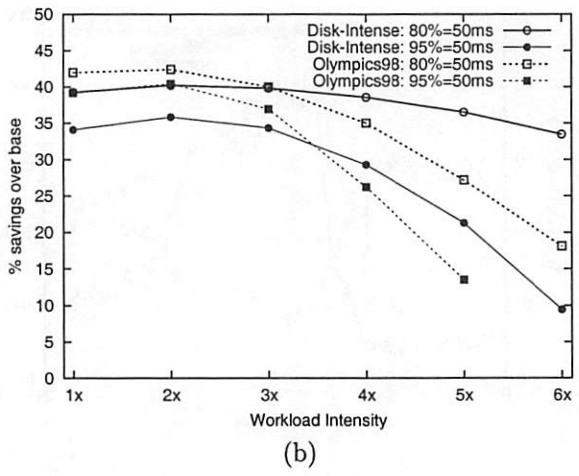


Figure 8: Energy savings from the combined policy. (a) Savings for all three workloads for the 90th-percentile response time goal of 50ms. (b) Savings range as percentile goal is varied for Disk-Intense and Olympics98.

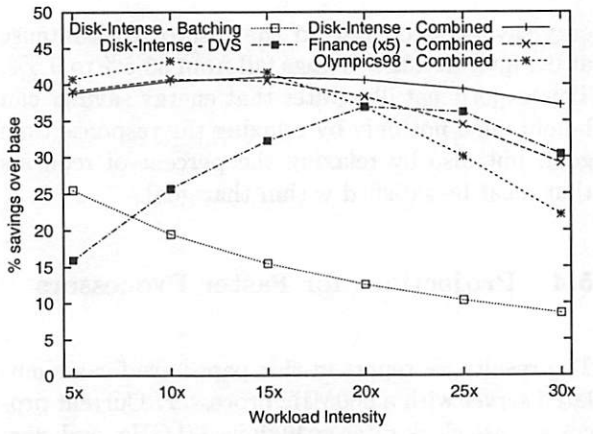


Figure 9: Energy savings from the combined policy for a 3.0 GHz processor for the 90th-percentile response time goal of 50ms.

model are presented in Table 4. These attributes are based loosely on reported power consumption for current high performance processors, with DVS support similar to that assumed for our 600MHz processor (Section 4.2). Note however that we have not validated our simulator against an actual 3.0 GHz processor, so the results in this section are intended only to establish the basic trend in effectiveness of our policies as CPU speed increases.

With faster processors, the savings from DVS (as a percentage of the total energy consumed) is likely to remain the same. The savings from request batching are likely to increase if the energy consumed during Deep Sleep is a smaller fraction of the idle power consumption. Indeed, this is true for most (if not all) Intel processors that are faster than the one we consider in this paper. Figure 9 shows the energy savings achieved by the combined policy on all three workloads and energy savings from DVS and request batching for the Disk-Intense workload using the 3.0 GHz processor model. These results confirm the expected trends described above. Calibrating and validating a new processor model in Salsa is a time intensive activity, and we have left a more complete evaluation of our policies on faster processors as future work.

5.5 Extensions to Other Environments

The results we report in this paper are for static workloads. Dynamic workloads introduce two new factors. First, requests take longer to service, causing significantly fewer periods where no requests are being processed. To handle dynamic requests, the batching policy we described in Section 3.1 will have to place the processor in Deep Sleep when all processes are blocked, as opposed to when no requests are being serviced. Second, dynamic requests can take significantly longer to process than static requests. Thus, the response time requirements must be relaxed beyond that we have used for static requests in order to permit the energy conservation policies to be applied. Furthermore, partitioning the QoS levels for static and dynamic requests will prevent the service times of one class from overshadowing the responsiveness metric used by the policies.

Several vendors have introduced "blade" systems that incorporate low-power processors in a spatially dense form factor [21]. While the policies described in this paper can be employed with low-power processors, the resulting large CPU energy savings may not translate into significant *system* energy savings since the CPU energy consumption may be a small component of the system energy in these systems. However, when such systems are clustered, policies that target server clusters by switching off entire systems become applicable [4, 19], and can be used in conjunction with the single–node policies described in this paper.

In earlier work [5], we studied combinations of cluster–level and single–node power management policies and found that fine–grained single–node energy management techniques can be very complimentary to typical cluster–level energy management mechanisms, which tend to be more coarse–grained in their effect on energy consumption and system performance. In addition, frequent powering off of systems may adversely affect the reliability of certain components, particularly disk drives, making single–node power management techniques more desirable for achieving energy savings from small or short term fluctuations in workload. Finally, single–node energy management policies are necessary for web sites that do not receive sufficient traffic to require multiple servers.

6 Related Work

Policies to reduce CPU energy consumption using dynamic voltage scaling have been widely studied. Early work used the CPU utilization over a recent set of intervals as a predictor of future system load, and then set the processor voltage and frequency for the next time internal so as to handle this load [9, 23]. Flautner et. al. explored a DVS policy that sets CPU speed on a per–task basis rather than for time intervals [7]. Lorch et. al. describe PACE, an algorithm to optimize task-based DVS policies when task completion times cannot be accurately predicted [14]. While these techniques perform well for desktop workloads, they are unsuitable for server environments with many concurrent tasks. Our DVS policy keeps track of the response time of individual requests ("tasks"), but employs a feedback–driven control framework to meet responsiveness requirements in the aggregate, instead of for individual tasks.

A number of recent papers have considered techniques for reducing energy consumption of web servers in large data centers. Pinheiro et. al. [19] proposed a simple policy for managing energy use in server clusters by powering machines on and off. Chase et. al. have employed this mechanism in the context of an economic framework in which web sites "bid" on resources based on their current workload [4]. In prior work, we explored the combination of the power-on/power-off technique with voltage scaling [5] to reduce energy consumption for a cluster of servers. The energy management techniques proposed in this paper are applicable to individual web server systems, and therefore complement efforts to manage energy for web server clusters.

Several researchers have developed tools for simulating the power consumption of computer systems. Brooks et. al. have developed Wattch, a microprocessor power analysis tool based on a microarchitecture simulator [3]. Flinn and Satyanarayanan describe PowerScope, a tool for profiling the energy usage of applications [8]. Our Salsa simulator is substantially faster, because it is targeted specifically for web serving workloads.

7 Conclusions

This paper describes three policies for reducing the energy consumption of server processors during web serving. The first policy uses dynamic voltage scaling (DVS), an existing energy-saving mechanism, but employs it in a feedback–driven control framework. The second policy is based on request batching, a new mechanism we introduce in this paper, that groups requests under periods of low workload intensity and executes them in batches, otherwise placing the processor in a Deep Sleep mode to conserve energy. These two policies target complementary ranges of workload intensities. While DVS is most effective for moderately intense workloads, request batching saves energy for light workloads. The third policy leverages both request batching and DVS mechanisms in a policy that saves energy across a wide range of workload intensities. All three policies conserve energy while maintaining a given quality of service level, as defined by a percentile–level response time.

The policies are evaluated using Salsa, a web serving simulator that has been validated for both energy and response times against an actual web server. The effectiveness of the policies are measured using three day–long workloads derived from real web server systems. When 90% of the requests must be serviced within a response time of 50ms, the DVS policy saves from 8.7% to 38% of the CPU energy used by the base system. The request batching policy provides from 3.1% to 27% savings for the same response–time requirement. However, the two polices provide these savings for different regions of the workloads. The combined policy is effective for all three workloads across a broad range of intensities, saving from 17% to 42% of the CPU energy.

Acknowledgements

We wish to thank our shepherd, Amin Vahdat, and the anonymous reviewer for many valuable comments and insights which have improved the quality of this paper.

References

[1] Alteon WebSystems Inc. Gigabit ethernet/PCI network interface card: Host/NIC software interface definition, 1999.

[2] P. Bohrer, E. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case for power management in web servers. In Robert Graybill and Rami Melhem, editors, *Power-Aware Computing*. Kluwer/Plenum Series in Computer Science, January 2002.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture*, pages 83–94, 2000.

[4] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[5] E. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proceedings of the Workshop on Power-Aware Computing Systems*, February 2002.

[6] Puget Sound Energy. Schedule 449: Retail Wheeling Service, 2002. Also see 'Schedule 448: Power Supplier Choice'.

[7] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[8] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 2–10, 1999.

[9] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the ACM International Conference on Mobile Computing and Networking*, pages 13–25, November 1995.

[10] Intel Corporation. Wired for management baseline version 2.0, 1999.

[11] Intel Corporation. Intel pentium 4 processor datasheet. Order Number: 249887-003, April 2002.

[12] The IRCache project. http://www.ircache.net/.

[13] A. Iyengar, M. Squillante, and L. Zhang. Analysis and characterization of large-scale web server access patterns and performance. *World Wide Web*, 2(1-2):85–100, June 1999.

[14] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

[15] Mesquite Software Inc. *CSIM18 Simulation Engine*, 1994.

[16] J. Mitchell-Jackson. Energy needs in an internet economy: A closer look at data centers. Master's thesis, University of California, Berkeley, July 2001.

[17] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *SIGMETRICS First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[18] K. Nowka. Private communication. IBM Research, Austin, TX.

[19] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[20] R. Rajamony and M. Elnozahy. Measuring client perceived response times on the WWW. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, March 2001.

[21] RLX Technologies Inc. Serverblade product description. http://www.rlx.com/products-/products_blades.php, 2002.

[22] BS Web Services. SLA. http://www.bsws.de/en-/products/sla-ahp.shtml.

[23] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, California, U.S., 1994.

SkipNet: A Scalable Overlay Network with Practical Locality Properties

Nicholas J.A. Harvey*† Michael B. Jones* Stefan Saroiu† Marvin Theimer* Alec Wolman*

Abstract: *Scalable overlay networks such as Chord, CAN, Pastry, and Tapestry have recently emerged as flexible infrastructure for building large peer-to-peer systems. In practice, such systems have two disadvantages: They provide no control over where data is stored and no guarantee that routing paths remain within an administrative domain whenever possible. SkipNet is a scalable overlay network that provides controlled data placement and guaranteed routing locality by organizing data primarily by string names. SkipNet allows for both fine-grained and coarse-grained control over data placement: Content can be placed either on a pre-determined node or distributed uniformly across the nodes of a hierarchical naming subtree. An additional useful consequence of SkipNet's locality properties is that partition failures, in which an entire organization disconnects from the rest of the system, can result in two disjoint, but well-connected overlay networks.*

1 Introduction

Scalable overlay networks, such as Chord [27], CAN [21], Pastry [23], and Tapestry [32], have recently emerged as flexible infrastructure for building large peer-to-peer systems. A key function that these networks enable is a distributed hash table (DHT), which allows data to be uniformly diffused over all the participants in the peer-to-peer system.

While DHTs provide nice load balancing properties, they do so at the price of controlling where data is stored. This has at least two disadvantages: Data may be stored far from its users and it may be stored outside the administrative domain to which it belongs. This paper introduces SkipNet, a distributed generalization of Skip Lists [20], adapted to meet the goals of peer-to-peer systems. SkipNet is a scalable overlay network that supports traditional overlay functionality as well as two locality properties that we refer to as *content locality* and *path locality*.

Content locality refers to the ability to either explicitly place data on specific overlay nodes or distribute it across nodes within a given organization. Path locality refers to the ability to guarantee that message traffic between two overlay nodes within the same organization is routed within that organization only.

*Microsoft Research, Microsoft Corporation, Redmond, WA. {nickhar, mbj, theimer, alecw}@microsoft.com

†Department of Computer Science and Engineering, University of Washington, Seattle, WA. {nickhar, tzoompy}@cs.washington.edu

Content and path locality provide a number of advantages for data retrieval, including improved availability, performance, manageability, and security. For example, nodes can store important data within their organization (content locality) and nodes will be able to reach their data through the overlay even if the organization becomes disconnected from the rest of the Internet (path locality). Storing data near the clients that use it also yields performance benefits. Placing content onto a specific overlay node—or a well-defined set of overlay nodes—enables provisioning of those nodes to reflect demand. Content placement also allows administrative control over issues such as scheduling maintenance for machines storing important data, thus improving manageability.

Content locality can improve security by allowing one to control the administrative domain in which data resides. Even when encrypted and digitally signed, data stored on an arbitrary overlay node outside the organization is susceptible to denial of service (DoS) attacks as well as traffic analysis. Although other techniques for improving the resiliency of DHTs to DoS attacks exist [3], content locality is a simple, zero-overhead technique.

Path locality provides additional security benefits to an overlay that supports content locality. Although some overlay designs [4] are likely to keep routing messages within an organization most of the time, none *guarantee* path locality. For example, without such a guarantee the route from *explorer.ford.com* to *mustang.ford.com* could pass through *camaro.gm.com*, a scenario that people at *ford.com* might prefer to prevent. With path locality, nodes requesting data within their organization traverse a path that never leaves the organization.

Controlling content placement is in direct tension with the goal of a DHT, which is to uniformly distribute data across a system in an automated fashion. A significant contribution of this paper is the concept of *constrained load balancing*, which is a generalization that combines these two notions: Data is uniformly distributed across a well-defined subset of the nodes in a system, such as all nodes in a single organization, all nodes residing within a given building, or all nodes residing within one or more data centers.

SkipNet supports efficient message routing between overlay nodes, content placement, path locality, and constrained load balancing. It does so by employing two separate, but related address spaces: a string name ID space as well as a numeric ID space. Node names and content identifier strings are mapped directly into the name ID space, while hashes of the node names and content identifiers are mapped into the numeric ID space. A single set of routing pointers on each overlay node enables efficient routing in either address space and a combination of routing in both address spaces provides the ability to do constrained load balancing.

A useful consequence of SkipNet's locality properties is resiliency against a common form of Internet failure. Because SkipNet clusters nodes according to their name ID ordering, nodes within a single organization gracefully survive failures that disconnect the organization from the rest of the Internet. In the case of uncorrelated, independent failures, SkipNet has similar resiliency to previous overlay networks [23, 27].

The basic SkipNet design, not including its enhancements to support constrained load balancing or network proximity-aware routing, has been concurrently and independently invented by Aspnes and Shah [1]. As described in Section 2, their work has a substantially different focus than our work and the two efforts are complementary to each other while still starting from the same underlying inspiration.

The rest of this paper is organized as follows: Section 2 describes related work, Section 3 describes SkipNet's basic design, Section 4 discusses SkipNet's locality properties, Section 5 presents enhancements to the basic design, Section 6 discusses design alternatives to SkipNet, Section 7 presents an experimental evaluation, and Section 8 concludes the paper.

2 Related Work

A large number of peer-to-peer overlay network designs have been proposed recently, such as CAN [21], Chord [27], Freenet [6], Gnutella [10], Pastry [23], Salad [9], Tapestry [32], and Viceroy [18]. SkipNet is designed to provide the same functionality as existing peer-to-peer overlay networks, and additionally to provide improved content availability through explicit control over content placement.

One key feature provided by systems such as CAN, Chord, Pastry, and Tapestry is scalable routing performance while maintaining a scalable amount of routing state at each node. By scalable routing paths we mean that the expected number of forwarding hops between any two communicating nodes is small with respect to the total number of nodes in the system. Chord, Pastry, and Tapestry scale with $\log N$, where N is the system size, while maintaining $\log N$ routing state at each overlay node. CAN scales with $D \cdot N^{1/D}$, where D is a parameter with a typical value of 6, while maintaining an amount of per-node routing state proportional to D.

A second key feature of these systems is that they are able to route to destination addresses that do not equal the address of any existing node. Each message is routed to the node whose address is "closest" to that specified in the destination field of a message; we interchangeably use the terms "route" and "search" to mean routing to the closest node to the specified destination. This feature enables implementation of a distributed hash table (DHT) [11], in which content is stored at an overlay node whose node ID is closest to the result of applying a collision-resistant hash function to that content's name (i.e. consistent hashing [15]).

Distributed hash tables have been used, for instance, in constructing the PAST [24] and CFS [7] distributed filesystems, the Overlook [29] scalable name service, the Squirrel [14] cooperative web cache, and scalable application-level multicast [5, 25, 22]. For most of these systems, if not all of them, the overlay network on which they were designed can easily be substituted with SkipNet.

SkipNet has a fundamental philosophical difference from existing overlay networks, such as Chord and Pastry, whose goal is to implement a DHT. The basic philosophy of systems like Chord and Pastry is to diffuse content randomly throughout an overlay in order to obtain uniform, load-balanced, peer-to-peer behavior. The basic philosophy of SkipNet is to enable systems to preserve useful content and path locality, while still enabling load balancing over constrained subsets of participating nodes.

This paper is not the first to observe that locality properties are important in peer-to-peer systems. Keleher et al. [16] make two main points: locality is a good thing, and DHTs destroy locality. Vahdat et al. [30] raises the locality issue as well. SkipNet addresses this problem directly: By using names rather than hashed identifiers to order nodes in the overlay, natural locality based on the names of objects is preserved. Furthermore, by arranging content in name order rather than dispersing it, efficient operations on ranges of names are possible in SkipNet, enabling, among other things, constrained load balancing.

Aspnes and Shah [1] have independently invented the same basic data structure that defines a SkipNet, which they call a Skip Graph. Beyond that, they investigate questions that are mostly orthogonal to those addressed in this paper. In particular, they describe and analyze different search and insertion algorithms and they focus on formal characterization of Skip Graph invariants. In contrast, our work is focused primarily on the content and path locality properties of the design, and we describe several extensions that are important in building a practical system: network proximity-aware routing is obtained by means of two auxiliary routing tables, and constrained load balancing is supported through a combination of searches in both the string name and numeric address spaces that SkipNet defines.

3 Basic SkipNet Structure

In this section, we introduce the basic design of SkipNet. We present the SkipNet architecture, including how to route in SkipNet, and how to join and leave a SkipNet.

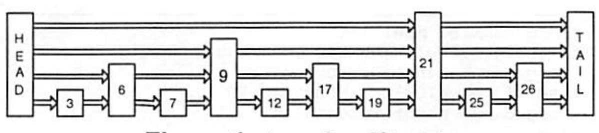


Figure 1. *A perfect Skip List.*

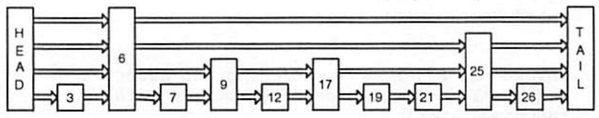


Figure 2. *A probabilistic Skip List.*

3.1 Analogy to Skip Lists

A Skip List, first described in Pugh [20], is a dictionary data structure typically stored in-memory. A Skip List is a sorted linked list in which some nodes are supplemented with pointers that skip over many list elements. A "perfect" Skip List is one where the height of the i^{th} node is the exponent of the largest power-of-two that divides i. Figure 1 depicts a perfect Skip List. Note that pointers at level h have length 2^h (i.e., they traverse 2^h nodes). A perfect Skip List supports searches in $O(\log N)$ time.

Because it is prohibitively expensive to perform insertions and deletions in a perfect Skip List, Pugh suggests a probabilistic scheme for determining node heights while maintaining $O(\log N)$ searches with high probability. Briefly, each node chooses a height such that the probability of choosing height h is $1/2^h$. Thus, with probability $1/2$ a node has height 1, with probability $1/4$ it has height 2, etc. Figure 2 depicts a probabilistic Skip List.

Whereas Skip Lists are an in-memory data structure that is traversed from its head node, we desire a data structure that links together distributed computer nodes and supports traversals that may start from any node in the system. Furthermore, because peers should have uniform roles and responsibilities in a peer-to-peer system, we desire that the state and processing overhead of all nodes be roughly the same. In contrast, Skip Lists maintain a highly variable number of pointers per data record and experience a substantially different amount of traversal traffic at each data record.

3.2 The SkipNet Structure

The key idea we take from Skip Lists is the notion of maintaining a sorted list of all data records as well as pointers that "skip" over varying numbers of records. We transform the concept of a Skip List to a distributed system setting by replacing data records with computer nodes, using the string *name IDs* of the nodes as the data record keys, and forming a ring instead of a list. The ring must be doubly-linked to enable path locality, as is explained in Section 3.3.

Rather than having nodes store a highly variable number of pointers, as in Skip Lists, each SkipNet node stores $2 \log N$ pointers, where N is the number of nodes in the overlay system. Each node's set of pointers is called its *routing table*, or R-Table, since the pointers are used to

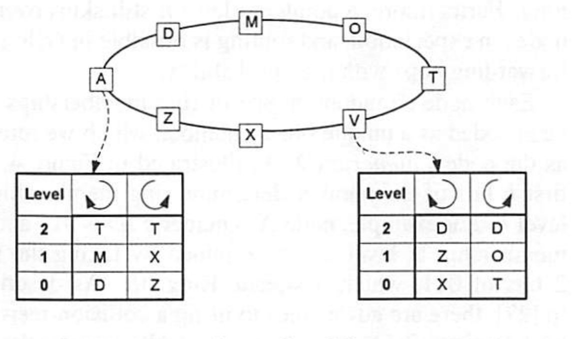


Figure 3. *SkipNet nodes ordered by name ID. Routing tables of nodes A and V are shown.*

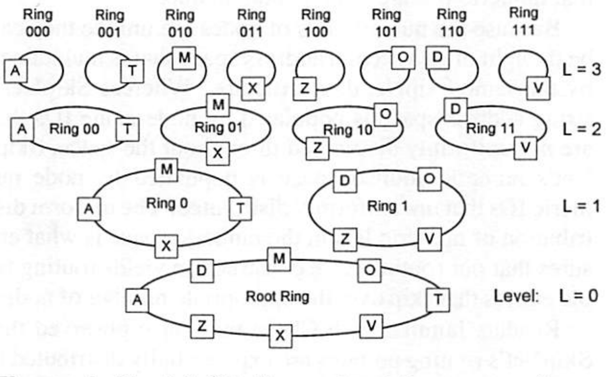


Figure 4. *The full SkipNet routing infrastructure for an 8 node system, including the ring labels.*

route message traffic between nodes. The pointers at level h of a given node's routing table point to nodes that are roughly 2^h nodes to the left and right of the given node. Figure 3 depicts a SkipNet containing eight nodes and shows the routing table pointers that nodes A and V maintain.

The SkipNet in Figure 3 is a "perfect" SkipNet: each level h pointer traverses exactly 2^h nodes. Figure 4 depicts the same SkipNet of Figure 3, arranged to show all node interconnections at every level simultaneously. All nodes are connected by the *root ring* formed by each node's pointers at level 0. The pointers at level 1 point to nodes that are 2 nodes away and hence the overlay nodes are implicity divided into two disjoint rings. Similarly, pointers at level 2 form four disjoint rings of nodes, and so forth. Note that rings at level $h + 1$ are obtained by splitting a ring at level h into two disjoint sets, each ring containing every second member of the level h ring.

Maintaining a perfect SkipNet in the presence of insertions and deletions is impractical, as is the case with perfect Skip Lists. To facilitate efficient insertions and deletions, we derive a probabilistic SkipNet design. Each ring at level h is split into two rings at level $h+1$ by having each node randomly and uniformly choose to which of the two rings it belongs. With this probabilistic scheme, insertion/deletion of a node only affects two other nodes in each ring to which the node has randomly chosen to belong. Furthermore, a pointer at level h still skips over 2^h nodes in expectation, and routing is possible in $O(\log N)$ forwarding hops with high probability.

Each node's random choice of ring memberships can be encoded as a unique binary number, which we refer to as the node's *numeric ID*. As illustrated in Figure 4, the first h bits of the number determine ring membership at level h. For example, node X's numeric ID is 011 and its membership at level 2 is determined by taking the first 2 bits of 011, which designate Ring 01. As described in [27], there are advantages to using a collision-resistant hash (such as SHA-1) of the node's DNS name as the numeric ID. The SkipNet design does not require the use of hashing to generate nodes' numeric IDs; we only require that numeric IDs are random and unique.

Because the numeric IDs of nodes are unique they can be thought of as a second address space that is maintained by the same SkipNet data structure. Whereas SkipNet's string address space is populated by node name IDs that are *not* uniformly distributed throughout the space, SkipNet's numeric address space is populated by node numeric IDs that *are* uniformly distributed. The uniform distribution of numeric IDs in the numeric space is what ensures that our routing table construction yields routing table entries that skip over the appropriate number of nodes.

Readers familiar with Chord may have observed that SkipNet's routing pointers are exponentially distributed in a manner similar to Chord's: The pointer at level h hops over 2^h nodes in expectation. The fundamental difference is that Chord's routing pointers skip over 2^h nodes in the numeric space. In contrast SkipNet's pointers, when considered from level 0 upward, skip over 2^h nodes in the name ID space and, when considered from the top level downward, skip over 2^h nodes in the numeric ID space. Chord guarantees $O(\log N)$ routing and node insertion performance by uniformly distributing node identifiers in its numeric address space. SkipNet guarantees $O(\log N)$ performance of node insertion and routing in both the name ID and numeric ID spaces by uniformly distributing numeric IDs and leveraging the sorted order of name IDs.

3.3 Routing by Name ID

Routing/searching by name ID in SkipNet is based on the same basic principle as searching in Skip Lists: Follow pointers that route closest to the intended destination. At each node, a message will be routed along the highest-level pointer that does not point past the destination value. Routing terminates when the message arrives at a node whose name ID is closest to the destination. Figure 5 presents this algorithm in pseudocode.

```
SendMsg(nameID, msg) {
  if( LongestPrefix(nameID,localNode.nameID)==0 )
    msg.dir = RandomDirection();
  else if( nameID<localNode.nameID )
    msg.dir = counterClockwise;
  else
    msg.dir = clockwise;
  msg.nameID = nameID;
  RouteByNameID(msg);
}

// Invoked at all nodes (including the source and
// destination nodes) along the routing path.
RouteByNameID(msg) {
  // Forward along the longest pointer
  // that is between us and msg.nameID.
  h = localNode.maxHeight;
  while (h >= 0) {
    nbr = localNode.RouteTable[msg.dir][h];
    if (LiesBetween(localNode.nameID, nbr.nameID,
                    msg.nameID, msg.dir)) {
      SendToNode(msg, nbr);
      return;
    }
    h = h - 1;
  }
  // h<0 implies we are the closest node.
  DeliverMessage(msg.msg);
}
```

Figure 5. *Algorithm for routing by name ID in SkipNet.*

Since nodes are ordered by name ID along each ring and a message is never forwarded past its destination, all nodes encountered during routing have name IDs between the source and the destination. Thus, when a message originates at a node whose name ID shares a common prefix with the destination, all nodes traversed by the message have name IDs that share that same prefix. Because rings are doubly-linked, this scheme can route using either right or left pointers depending upon whether the source's name ID is smaller or greater than the destination's. The key observation of this scheme is that routing by name ID traverses only nodes whose name IDs share a non-decreasing prefix with the destination ID.

If the source name ID and the destination name ID share no common prefix, a message can be routed in either direction. For the sake of fairness, we randomly pick a direction so that nodes whose name IDs are near the middle of the sorted ordering do not get a disproportionately large share of the forwarding traffic.

The number of message hops when routing by name ID is $O(\log N)$ with high probability. For a proof of this, see [12].

3.4 Routing by Numeric ID

It is also possible to route messages efficiently to a given numeric ID. In brief, the routing operation begins by examining nodes in the level 0 ring until a node is found whose numeric ID matches the destination numeric ID in the first digit. At this point the routing operation jumps up to this node's level 1 ring, which also contains the destination node. The routing operation then examines nodes in this level 1 ring until a node is found whose numeric ID matches the destination numeric ID in the second digit. As before, we know that this node's level 2 ring must also contain the destination node, and thus the routing operation proceeds in this level 2 ring.

This procedure repeats until we cannot make any more progress — we have reached a ring at some level h such that none of the nodes in that ring share $h+1$ digits with the destination numeric ID. We must now deterministically choose one of the nodes in this ring to be the desti-

```
// Invoked at all nodes (including the source and
// destination nodes) along the routing path.
// Initially:
//    msg.ringLvl = -1
//    msg.startNode = msg.bestNode = null
//    msg.finalDestination = false
RouteByNumericID(msg) {
  if (msg.numID == localNode.numID ||
      msg.finalDestination) {
    DeliverMessage(msg.msg);
    return;
  }

  if (localNode == msg.startNode) {
    // Done traversing current ring.
    msg.finalDestination = true;
    SendToNode(msg.bestNode);
    return;
  }

  h = CommonPrefixLen(msg.numID, localNode.numID);
  if (h > msg.ringLvl) {
    // Found a higher ring.
    msg.ringLvl = h;
    msg.startNode = msg.bestNode = localNode;
  } else if ( abs(localNode.numID - msg.numID) <
            abs(msg.bestNode.numID - msg.numID)) {
    // Found a better candidate for current ring.
    msg.bestNode = localNode;
  }

  // Forward along current ring.
  nbr = localNode.RouteTable[clockWise][msg.ringLvl];
  SendToNode(nbr);
}
```

Figure 6. *Algorithm to route by numeric ID in SkipNet*

nation node. Our algorithm defines the destination node to be the node whose numeric ID is numerically closest to destination numeric ID amongst all nodes in this highest ring. Figure 6 presents this algorithm in pseudocode.

As an example, imagine that the numeric IDs in Figure 4 are 4 bits long and that node Z's ID is 1000 and node O's ID is 1001. If we want to route a message from node A to destination 1011 then A will first forward the message to node D because D is in ring 1. D will then forward the message to node O because O is in ring 10. O will forward the message to Z because it is not in ring 101. Z will forward the message onward around the ring (and hence back) to O for the same reason. Since none of the members of ring 10 belong to ring 101, node O will be picked as the final message destination because its numeric ID is closest to 1011 of all ring 10 members.

The number of message hops when routing by numeric ID is $O(\log N)$ with high probability. For a proof of this, see [12].

Some intuition for why SkipNet can support efficient routing by both name ID and numeric ID with the same data structure is illustrated in Figure 4. Note that the root ring, at the bottom, is sorted by name ID and, collectively, the top-level rings are sorted by numeric ID. For any given node, the SkipNet rings to which it belongs precisely form a Skip List. Thus efficient searches by name ID are possible. Furthermore, if you construct a trie on all nodes' numeric IDs, the nodes of the resulting trie would be in one-to-one correspondence with the SkipNet rings. This suggests that efficient searches by numeric ID are also possible.

```
InsertNode(nameID, numID) {
  msg = new JoinMessage();
  msg.operation = findTopLevelRing;
  RouteByNumericID(numID, msg);
}

DeliverMessage(msg) {
  ...
  else if (msg.operation == findTopLevelRing) {
    msg.ringLvl =
        CommonPrefix(localNode.numID, msg.numID);
    msg.ringNbrClockWise = new Node[msg.ringLvl];
    msg.ringNbrCClockWise = new Node[msg.ringLvl];
    msg.doInsertions = false;
    CollectRingInsertionNeighbors(msg);
  }
  else ...
}

// Invoked at every intermediate routing hop.
CollectRingInsertionNeighbors(msg) {
  if (msg.doInsertions) {
    InsertIntoRings(msg.ringNbrClockWise,
                    msg.ringNbrCClockWise);
    return;
  }

  while (msg.ringLvl >= 0) {
    nbr = localNode.RouteTable[clockWise][msg.ringLvl];
    if (LiesBetween(localNode.nameID, msg.nameID,
            nbr.nameID, clockWise)) {
      // Found an insertion neighbor.
      msg.ringNbrClockWise[msg.ringLvl] = nbr;
      msg.ringNbrCClockWise[msg.ringLvl] = localNode;
      msg.ringLvl = msg.ringLvl-1;
    } else {
      // Keep looking
      SendToNode(msg, nbr);
      return;
    }
  }

  msg.doInsertions = true;
  SendToNode(msg, msg.joiningNode);
}
```

Figure 7. *Algorithm to insert a SkipNet node.*

3.5 Node Join and Departure

To join a SkipNet, a newcomer must first find the top-level ring that corresponds to the newcomer's numeric ID. This amounts to routing a message to the newcomer's numeric ID, as described in Section 3.4.

The newcomer then finds its neighbors in this top-level ring, using a search by name ID within this ring only. Starting from one of these neighbors, the newcomer searches for its name ID at the next lower level and thus finds its neighbors at this lower level. This process is repeated for each level until the newcomer reaches the root ring. For correctness, the existing nodes only point to the newcomer after it has joined the root ring; the newcomer then notifies its neighbors in each ring that it should be inserted next to them. Figure 7 presents this algorithm in pseudocode.

As an example, imagine inserting node O into the SkipNet of Figure 4. Node O initiates a search by numeric ID for its own ID (101) and the resulting insertion message ends up at node Z in ring 10 since that is the highest non-empty ring that shares a prefix with node O's numeric ID. Since Z is the only node in ring 10, Z concludes that it is both the clockwise and counter-clockwise neighbor of node O in this ring.

In order to find node O's neighbors in the next lower ring (ring 1), node Z forwards the insertion message to node D. Node D then concludes that D and V are the neighbors of node O in ring 1. Similarly, node D forwards the insertion message to node M in the root ring, who concludes that node O's level 0 neighbors must be M and T. The insertion message is returned to node O, who then instructs all of its neighbors to insert it into the rings.

The key observation for this algorithm's efficiency is that a newcomer searches for its neighbors at a certain level only after finding its neighbors at all higher levels. As a result, the search by name ID will traverse only a few nodes within each ring to be joined: The range of nodes traversed at each level is limited to the range between the newcomer's neighbors at the next higher level. Therefore, with high probability, a node join in SkipNet will traverse $O(\log N)$ hops (for a proof see [12]).

The basic observation in handling node departures is that SkipNet can route correctly as long as the bottom level ring is maintained. All pointers but the level 0 ones can be regarded as routing optimization hints, and thus are not necessary to maintain routing protocol correctness. Therefore, like Chord and Pastry, SkipNet maintains and repairs the upper-level ring memberships by means of a background repair process. In addition, when a node voluntarily departs from the SkipNet, it can proactively notify all of its neighbors to repair their pointers immediately.

To maintain the root ring correctly, each SkipNet node maintains a leaf set that points to additional nodes along the root ring, for redundancy. In our current implementation we use a leaf set size of 16, just as Pastry does.

4 Useful Locality Properties of SkipNet

In this section we discuss some of the useful locality properties that SkipNet is able to provide, and their consequences.

4.1 Content and Routing Path Locality

Given the basic structure of SkipNet, describing how SkipNet supports content and path locality is straightforward. Incorporating a node's name ID into a content name guarantees that the content will be hosted on that node. As an example, to store a document *doc-name* on the node *john.microsoft.com*, naming it *john.microsoft.com/doc-name* is sufficient.

SkipNet is oblivious to the naming convention used for nodes' name IDs. Our simulations and deployments of SkipNet use DNS names for name IDs, after suitably reversing the components of the DNS name. In this scheme, *john.microsoft.com* becomes *com.microsoft.john*, and thus all nodes within *microsoft.com* share the *com.microsoft* prefix in their name IDs. This yields path locality for organizations in which all nodes share a single DNS suffix (and hence share a single name ID prefix).

4.2 Constrained Load Balancing

As mentioned in the Introduction, SkipNet supports Constrained Load Balancing (CLB). To implement CLB, we divide a data object's name into two parts: a part that specifies the set of nodes over which DHT load balancing should be performed (the *CLB domain*) and a part that is used as input to the DHT's hash function (the *CLB suffix*). In SkipNet the special character '!' is used as a delimiter between the two parts of the name.

For example, suppose we stored a document using the name *msn.com/DataCenter!TopStories.html*. The CLB domain indicates that load balancing should occur over all nodes whose names begin with the prefix *msn.com/DataCenter*. The CLB suffix, *TopStories.html*, is used as input to the DHT hash function, and this determines the specific node within *msn.com/DataCenter* on which the document will be placed. Note that storing a document with CLB results in the document being placed on precisely one node within the CLB domain (although it would be possible to store replicas on other nodes). If numerous other documents were also stored in the *msn.com/DataCenter* CLB domain, then the documents would be uniformly distributed across all nodes in that domain.

To search for a data object that has been stored using CLB, we first search for any node within the CLB domain using search by name ID. To find the specific node within the domain that stores the data object, we perform a search by numeric ID within the CLB domain for the hash of the CLB suffix.

The search by name ID is unmodified from the description in Section 3.3, and takes $O(\log N)$ message hops. The search by numeric ID is constrained by a name ID prefix and thus at any level must effectively step through a doubly-linked list rather than a ring. Upon encountering the right boundary of the list (as determined by the name ID prefix boundary), the search must reverse direction in order to ensure that no node is overlooked. Reversing directions in this manner affects the performance of the search by numeric ID by at most a factor of two, and thus $O(\log N)$ message hops are required in total.

Note that both traditional system-wide DHT semantics as well as explicit content placement are special cases of constrained load balancing: system-wide DHT semantics are obtained by placing the '!' hashing delimiter at the beginning of a document name. Omission of the hashing delimiter and choosing the name of a data object to have a prefix that matches the name of a particular SkipNet node will result in that data object being placed on that SkipNet node.

Constrained load balancing can be performed over any naming subtree of the SkipNet but not over an arbitrary subset of the nodes of the overlay network. Another limitation is that CLB domain is encoded in the name of a data object. Thus, transparent remapping to a different load balancing domain is not possible.

4.3 Fault Tolerance

Previous studies [17, 19] indicate that network connectivity failures in the Internet today are due primarily to Border Gateway Protocol (BGP) misconfigurations and faults. Other hardware, software and human failures play a lesser role. As a result, node failures in overlay networks are not independent; instead, nodes belonging to the same organization or AS tend to fail together. We consider both correlated and independent failure cases in this section.

4.3.1 Independent Failures

SkipNet's tolerance to uncorrelated, independent failures is much the same as previous overlay designs' (e.g., Chord and Pastry), and is achieved through similar mechanisms. The key observation in failure recovery is that maintaining correct neighbor pointers in the level 0 ring is enough to ensure correct functioning of the overlay. Since each node maintains a leaf set of 16 neighbors at level 0, the level 0 ring pointers can be repaired by replacing them with the leaf set entries that point to the nearest live nodes following the failed node. The live nodes in the leaf set may be contacted to repopulate the leaf set fully.

SkipNet also employs a background stabilization mechanism that gradually updates all necessary routing table entries when a node fails. Any query to a live, reachable node will still succeed during this time; the stabilization mechanism simply restores optimal routing.

4.3.2 Failures along Organization Boundaries

In previous peer-to-peer overlay designs [21, 27, 23, 32], node placement in the overlay topology is determined by a randomly chosen numeric ID. As a result, nodes within a single organization are placed uniformly throughout the address space of the overlay. While a uniform distribution facilitates the $O(\log N)$ routing performance of the overlay it makes it difficult to control the effect of physical link failures on the overlay network. In particular, the failure of a inter-organizational network link may manifest itself as multiple, scattered link failures in the overlay. Indeed, it is possible for each node within a single organization that has lost connectivity to the Internet to become disconnected from the entire overlay and from all other nodes within the organization. Section 7.4 reports experimental results that confirm this observation.

Since SkipNet name IDs tend to encode organizational membership, and nodes with common name ID prefixes are contiguous in the overlay, failures along organization boundaries do not completely fragment the overlay, but instead result in ring segment partitions. Consequently, a significant fraction of routing table entries of nodes within the disconnected organization still point to live nodes within the same network partition. This property allows SkipNet to gracefully survive failures along organization boundaries. Furthermore, the disconnected organization's SkipNet segment can be efficiently re-merged with the external SkipNet when connectivity is restored, as described in a related paper [13].

4.4 Security

In this section, we discuss some security consequences of SkipNet's content and path locality properties. Recent work [3] on improving the security of peer-to-peer systems has focused on certification of node identifiers and the use of redundant routing paths. The security advantages of content and path locality depend on an access control mechanism for creation of name IDs. SkipNet does not directly provide this mechanism but rather assumes that it is provided at another layer. Our use of DNS names for name IDs does provide this mechanism: Arbitrary nodes cannot create global DNS names containing the suffix of a registered organization without its permission.

Path locality allows SkipNet to guarantee that messages between two machines within a single administrative domain that uses a single name ID prefix will never leave the administrative domain. Thus, these messages are not susceptible to traffic analysis or denial-of-service attacks by machines located outside of the administrative domain. Furthermore, traffic that is internal to an organization is not susceptible to a Sybil attack [8] originating from a foreign organization: Creating an unbounded number of nodes outside *microsoft.com* will not allow the attacker to see any traffic internal to *microsoft.com*, nor allow the attacker to usurp control over documents placed specifically within *microsoft.com*.

In Chord, the nodes belonging to an administrative domain (for example, *microsoft.com*) are uniformly dispersed throughout the overlay. Thus, intercepting a significant portion of the traffic to *microsoft.com* may require that an attacker create a large number of nodes. In SkipNet, the nodes belonging to an administrative domain form a contiguous segment of the overlay. Thus, an attacker might attempt to target *microsoft.com* by creating nodes (for example, *microsofa.com*) that are adjacent to the target domain. Thus a security disadvantage of SkipNet is that it may be possible to target traffic between an administrative domain and the outside world with fewer attacking nodes than would be necessary in systems such as Chord. We believe that susceptibility to these kinds of attacks is a small price to pay in return for the benefits provided by path and content locality.

4.5 Range Queries

Since SkipNet's design is based on and inspired by Skip Lists, it inherits their functionality and flexibility in supporting efficient range queries. In particular, since nodes and data are stored in name ID order, documents sharing common prefixes are stored over contiguous ring segments. Performing range queries in SkipNet is therefore equivalent to routing along the corresponding ring segment. Because our current focus is on SkipNet's architecture and locality properties, we do not discuss the use of range queries for implementing various higher-level data query operators further in this paper.

5 SkipNet Enhancements

This section presents several optimizations and enhancements to the basic SkipNet design.

5.1 Sparse and Dense Routing Tables

The basic SkipNet design may be modified in order to improve routing performance. Thus far in our discussions, SkipNet numeric IDs consist of random binary digits. However, we can also use non-binary random digits, which changes the ring structure depicted in Figure 4, the number of pointers stored per node, and the expected routing cost. We denote the number of different possibilities for a digit by k; in the binary digit case, $k = 2$. If $k = 3$, the root ring of SkipNet remains a single ring, but there are now three level 1 rings, nine level 2 rings, etc. As k increases, the total number of pointers in the R-Table will decrease. Because there are fewer pointers, it will take more routing hops to get to any particular node. We call the routing table that results from this modification a *sparse R-Table* with parameter k.

It is also possible to build a *dense R-Table* by additionally storing $k - 1$ pointers to contiguous nodes at each level of the routing table and in both directions. In this case, the expected number of search hops decreases while the expected number of pointers at a node increases.

Increasing k makes the sparse R-Table sparser and the dense R-Table denser. The density parameter k and choice of sparse or dense construction can be used to control the amount routing state used by all SkipNet routing tables, and in Section 7 we examine the relationship between routing performance and the amount of routing table state maintained.

Implementing node join and departure in the case of sparse R-Tables requires no modification to our previous algorithms. For dense R-Tables, the node join message must traverse and gather information about at least $k - 1$ nodes in both directions in every ring containing the newcomer, before descending to the next ring. As before, node departure merely requires notifying every neighbor.

5.2 Duplicate Pointer Elimination

Two nodes that are neighbors in a ring at level h may also be neighbors in a ring at level $h + 1$. In this case, these two nodes maintain "duplicate" pointers to each other at levels h and $h + 1$. Intuitively, routing tables with more distinct pointers yield better routing performance than tables with fewer distinct pointers, and hence duplicate pointers reduce the effectiveness of a routing table. Replacing a duplicate pointer with a suitable alternative, such as the following neighbor in the higher ring, improves routing performance by a moderate amount (our experiments indicate improvements typically around 25%). Routing table entries adjusted in this fashion can only be used when routing by name ID since they violate the invariant that a node point to its closest neighbor on a ring, which is required for correct routing by numeric ID.

5.3 Incorporating Network Proximity for Routing by Name ID

In SkipNet, a node's neighbors are determined by a random choice of ring memberships (i.e., numeric IDs) and by the ordering of name IDs within those rings. Accordingly, the SkipNet overlay is constructed without direct consideration of the physical network topology, potentially hurting routing performance. For example, when sending a message from the node *saturn.com/nodeA* to the node *chrysler.com/nodeB*, both in the USA, the message might get routed through the intermediate node *jaguar.com/nodeC* in the UK. This would result in a much longer path than if the message had been routed through another intermediate node in the USA.

To deal with this problem, we introduce a second routing table called the *P-Table*, which is short for proximity table. The goal of the P-Table is to maintain routing in $O(\log N)$ hops, while also ensuring that each hop has low cost in terms of network latency. Our P-Table design is inspired by Pastry's proximity-aware routing tables [4]. To incorporate network proximity into SkipNet, the key observation is that any node that is roughly the right distance away in name ID space can be used as an acceptable routing table entry that will maintain the underlying $O(\log N)$ routing performance. For example, it doesn't matter whether a P-Table entry at level 3 points to the node that is exactly 8 nodes away or to one that is 7 or 9 nodes away; statistically the number of forwarding hops that messages will take will end up being the same. However, if the 7th or 9th node is nearby in network distance then using it as the P-Table entry can yield substantially better routing performance. In fact, the P-Table entry at level h can be anywhere between 2^h and 2^{h+1} nodes away while maintaining $O(\log N)$ routing performance.

To construct its P-Table, a node needs to locate a set of candidate nodes that are close in terms of network distance and whose name IDs are appropriately distributed around the root ring. Unlike Chord and Pastry, in SkipNet it is difficult to estimate distance along the root ring simply by looking at a candidate node's name ID. We solve this problem by observing that a node's basic routing table (the R-Table) conveniently divides the root ring into intervals of exponentially increasing size. Thus, two pointers at adjacent levels in the R-Table provide the name ID boundaries of a contiguous interval along the root ring. Given a node, we examine these intervals to determine which P-Table entry it is a candidate for. We discover candidate nodes that are nearby using a recursive process: we start at a nearby *seed node* and discover other nearby nodes by querying the P-Table of the seed node. Finally, we determine that two nodes are near each other by estimating the round-trip latency between them.

The following section provides a detailed description of the algorithm that a SkipNet node uses to construct its P-Table. In [12], we provide an informal analysis of the performance of the P-Table routing and P-Table construction algorithms.

5.3.1 P-Table Construction

When a node joins SkipNet it first constructs its R-Table. P-Table construction is then initiated by copying the entries of the R-Table to a separate list, where they are sorted by name ID and then duplicate entries are eliminated. Duplicates and out-of-order entries can arise in this list due to the probabilistic nature of constructing the R-Table.

The joining node then constructs a P-Table join message that contains the sorted list of endpoints: a list of j nodes defining $j-1$ intervals. The joining node sends this P-Table join message to a *seed node* – a node that should be nearby in terms of network distance.

Every node that receives a P-Table join message uses its own P-Table entries to fill in the intervals with "candidate" nodes. After filling in any possible intervals, the node checks whether any of the intervals are still empty. If so, the node forwards the join message, using its own P-Table entries, towards an unfilled interval. [12] explains why forwarding the join message to the the farthest unfilled interval from the joining node yields the smallest expected number of insertion forwarding hops. If all the intervals have at least one candidate, the node sends the completed join message back to the original joining node.

When the original node receives its own join message, it chooses one candidate node per interval as its P-Table entry. The choice between candidate nodes is performed by estimating the network latency to each candidate and choosing the closest node.

We now summarize a few remaining key details of P-Table construction. Since SkipNet can route either clockwise or counter-clockwise, the P-Table contains intervals that cover the address space in both directions from the joining node. Thus two join messages are sent from the same starting node.

The effectiveness of P-Table routing entries is dependent to a great extent on finding nearby nodes. The basis of this process is finding a good *seed node*. In our simulator, we implemented two strategies for locating a seed node. Our first strategy uses global knowledge from the simulator topology model to find the closest node in the entire system. The second and more realistic strategy is that we choose the seed node at random, and then run the P-Table join algorithm twice. We use the first run of the P-Table join algorithm to locate a nearby seed, and the second run to construct a better P-table based on the nearby seed. Section 7.6 summarizes a performance evaluation of these two approaches.

After the initial P-Table is constructed, SkipNet constantly tries to improve the quality of its P-Table entries, and adjusts to node joins and departures, by means of a periodic stabilization algorithm. The P-Table is updated periodically so that the P-Table segment endpoints accurately reflect the distribution of name IDs in the SkipNet, which may change over time. The periodic mechanism used to update P-Table entries is very similar to the initial construction algorithm presented above. One key difference between the update mechanism and the initial construction mechanism is that for update, the current P-Table entries are considered as candidate nodes in addition to the candidates returned by the P-Table join message. The other difference is that for update, the seed node is chosen as the best candidate from the existing P-Table. Finally, the P-Table entries may also be incrementally updated as node joins and departures are discovered through ordinary message traffic.

5.4 Incorporating Network Proximity for Routing by Numeric ID

We add a third routing table, the C-Table, to incorporate network proximity when searching by numeric ID. Constrained Load Balancing (CLB), because it involves searches by both name ID and numeric ID, takes advantage of both the P-Table and the C-Table. Because search by numeric ID as part of a CLB search must stay within the CLB domain, C-Table entries that step outside the domain cannot be used. When such an entry is encountered, the CLB search must revert to using the R-Table.

The C-Table has identical functionality and design to the routing table that Pastry maintains [23]. Further details of the C-Table data structure and construction process can be found in [12].

6 Design Alternatives

SkipNet's locality properties can be obtained to a limited degree by suitable extensions to existing overlay network designs. We explore several such extensions in this section. However, none of these design alternatives provides all of SkipNet's locality advantages.

The space of alternative design choices can be divided into three cases: Rely on the inherent locality properties of the underlying IP network and DNS naming instead of using an overlay network; use a single overlay network—possibly augmented—that supports locality properties; or use multiple overlay networks that provide locality by spanning different sets of member nodes.

6.1 IP routing and DNS naming

A simple alternative to SkipNet's content placement scheme is to route directly using IP after a DNS lookup. This approach would also arguably provide path locality since most organizations structure their internal networks in a path-local manner. However, discarding the overlay network also discards all of its advantages, including:

- Implicit support for DHTs, and in the case of SkipNet, support for constrained load balancing.
- Seamless reassignment of traffic to well-defined alternative nodes in the presence of node failures.
- Better support for higher level abstractions, such as application-level multicast [5, 25, 22] and load-aware replication [29].
- The ability to reach named destinations independent of the availability of the DNS name lookup service.

6.2 Single Overlay Networks

Existing overlays are based on DHTs and depend on random assignment of node IDs in order to obtain a uniform distribution of nodes within their address spaces. To support explicit content placement onto a particular node requires changing either node or data naming. One could name a node with the hash of the data object's name, or some portion of its name. This scheme effectively virtualizes overlay nodes so that each node joins the overlay once per data object.

The drawback of this solution is that separate routing tables are required for each local data object. This will result in a prohibitive cost whenever a single node needs to store more than a few hundred data objects due to the network traffic overhead of building and maintaining large numbers of routing table entries.

Alternatively, one could change object names to use a two-part naming scheme, much like in SkipNet, where content names consist of unique node addresses concatenated to local, node-relative, names. Although this approach supports content placement, it does not support *guaranteed* path locality nor constrained load balancing (including continued content locality in the event of failover to a neighbor node).

One might imagine providing path locality by adding routing constraints to messages, so that messages are not allowed to be forwarded outside of a given organizational boundary. Unfortunately, such constraints would also prevent routing from being consistent. That is, messages sent to the same destination ID from two different source nodes would not be guaranteed to end up at the same destination node.

An alternative to virtualizing node names would be to lengthen node IDs and partition them into separate, concatenated parts. For example, in a two-part scheme, node names would consist of two concatenated IDs and content names would also consist of two parts: a numeric ID value and a string name. The numeric ID would map to the first part of an overlay ID while the hash of the string name would map to the second part. The result is a static form of constrained load balancing: The numeric ID of a data object's name selects the DHT formed by all nodes sharing the same numeric ID and the string name determines which node to map to within the selected DHT. Furthermore, combining this approach with node virtualization provides explicit content placement.

This approach comes close to providing the same locality semantics as SkipNet: it provides explicit content placement, a static form of constrained load balancing, and path locality within each numeric ID domain. The major drawbacks of this approach are that the granularity of the hierarchy is frozen at the time of overlay creation by human decision; every layer of the hierarchy incurs an additional cost in the length of the numeric ID and in the size of the routing table that must be maintained; and the path locality guarantee is only with respect to boundaries in the static hierarchy.

6.3 Multiple Overlay Networks

Instead of using a single DHT-based overlay one might consider employing multiple overlays with different memberships. These multiple overlays can be arranged either as a static set of networks reflecting the desired locality requirements or as a dynamic set of overlays reflecting the participation of nodes in particular applications. In the static overlay case, a node could belong to just one of several alternative overlays, or belong to multiple overlays at different levels of a hierarchy.

In the case where each node belongs to only one of several overlays, one could imagine accessing other overlays by gateways. These gateways need not be a single point of failure if we give the backup gateway an appropriate neighboring numeric ID. One could either route directly to well-known gateways, or the gateways could organize an overlay network amongst themselves (imagine a overlay network of overlay networks). In either case, inter-domain routing requires serial traversal of the domain hierarchy, resulting in potentially large latencies when routing between domains.

If instead each node belonged to multiple overlays (for example, to a global overlay, an organization-wide overlay, and perhaps also a divisional or building-wide overlay), the associated overhead would correspondingly grow. Explicit content placement would still require extension of the overlay design. Furthermore, in this scheme, access to data that is constrained load balanced within a single overlay is not readily accessible to clients outside that overlay network, although it could be made so by introducing gateways in this design.

A final design alternative involving multiple overlays is to define an overlay network per application. This lets applications dynamically define the set of participating nodes, and thus ensure that application specific messages stay within this overlay. It does not provide any notion of locality within a subset of the overlay, and therefore fails to provide much of SkipNet's functionality, such as constrained load balancing.

In contrast, SkipNet provides explicit content placement, allows clients to dynamically define new DHTs over any name prefix scope, and guarantees path locality within any shared name prefix, all within a single shared infrastructure.

7 Experimental Evaluation

To understand and evaluate SkipNet's design and performance, we used a simple packet-level, discrete event simulator that counts the number of packets sent over a physical link and assigns either a unit hop count or a specified delay for each link, depending upon the topology used. It does not model either queuing delay or packet losses because modelling these would prevent simulation of large networks.

Our simulator implements three overlay network designs: Pastry, Chord, and SkipNet. The Pastry implementation is described in [23]. Our Chord implementation is based on the algorithm in [27], adapted to operate within our simulator. For our simulations, we run the Chord stabilization algorithm until no finger pointers need updating after all nodes have joined. We use two different implementations of SkipNet: a "basic" implementation that uses only the R-Table with duplicate pointer elimination, and a "full" implementation that includes the P-Table and C-Table as well. The full SkipNet implementation uses a sparse R-Table, and a dense P-Table with density parameter $k = 8$. For full SkipNet, we run two rounds of stabilization for P-Table entries before each experiment. In addition to the information provided below, we provide a complete specification of all the configuration parameters used in our simulation runs in [12].

All our experiments were run both on a Mercator topology [28] and a GT-ITM topology [31]. The Mercator topology has 102,639 nodes and 142,303 links. Each node is assigned to one of 2,662 Autonomous Systems (ASs). There are 4,851 links between ASs in the topology. The Mercator topology assigns a unit hop count for each link. All figures shown in this section are for the Mercator topology. The experiments based on the GT-ITM topology produced similar results.

7.1 Methodology

We measured the performance characteristics of lookups using the following evaluation criteria:

Relative Delay Penalty (RDP): The ratio of the latency of the overlay network path between two nodes to the latency of the IP-level path between them.

Physical network hops: The absolute length of the overlay path between two nodes, measured in IP-level hops.

Number of failed lookups: The number of unsuccessful lookup requests in the presence of failures.

We also model the presence of organizations within the overlay network; each participating node belongs to a single organization. The number of organizations is a parameter to the experiment, as is the total number of nodes in the overlay. For each experiment, the total number of client lookups is ten times the number of nodes in the overlay.

The format of the names of participating nodes is *org-name/node-name*. The format of data object names is *org-name/node-name/random-obj-name*. Therefore we assume that the "owner" of a particular data object will name it with the owner node's name followed by a node-local object name. In SkipNet, this results in a data object being placed on the owner's node; in Chord and Pastry, the object is placed on a node corresponding to the SHA-1 hash of the object's name. For constrained load balancing experiments we use data object names that include the '!' delimiter following the name of the organization.

We model organization sizes two ways: a uniform model and a Zipf-like model. In the Zipf-like model, the size of an organization is determined according to a distribution governed by $x^{-1.25} + 0.5$ and normalized to the total number of overlay nodes in the system. All other Zipf-like distributions mentioned in this section are defined in a similar manner.

We model three kinds of node locality: uniform, clustered, and Zipf-clustered. In the uniform model, nodes are uniformly spread throughout the overlay. In the clustered model, the nodes of an organization are uniformly spread throughout a single randomly chosen autonomous system. We ensure that the selected AS has at least 1/10-th as many core router nodes as overlay nodes. For Zipf-clustered, we place organizations within ASes, as before. However, the nodes of an organization are spread throughout its AS as follows: A "root" physical node is randomly placed within the AS and all overlay nodes are placed relative to this root, at distances modelled by a Zipf-like distribution. In this configuration most of the overlay nodes of an organization will be closely clustered together within their AS. This configuration is especially relevant to the Mercator topology, in which some ASes span large portions of the entire topology.

Data object names, and therefore data placement, are modelled similarly. In a uniform model, data names are generated by randomly selecting an organization and then a random node within that organization. In a clustered model, data names are generated by selecting an organization according to a Zipf-like distribution and then a random member node within that organization. For Zipf-clustered, data names are generated by randomly selecting an organization according to a Zipf-like distribution and then selecting a member node according to a Zipf-like distribution of its distance from the "root" node of the organization. Note that for Chord and Pastry, but not SkipNet, hashing spreads data objects uniformly among all overlay nodes in all of these three models.

For SkipNet, the actual node names used in our simulations may impact performance, so we used realistic distributions for both host names and organization names. Our distribution of organization names was derived from a list of 5,608 unique organizations which had at least one peer participating in Gnutella in March 2001 [26]. The host name distribution was obtained from a list of 177,000 internal host names in use at Microsoft Corporation.

We model locality of data access by specifying what fraction of all data lookups will be forced to request data local to the requestor's organization. Finally, we model system behavior under Internet-like failures and study document availability within a disconnected organization. We simulate domain isolation by failing the links connecting the organization's AS to the rest of the network.

Each experiment is run ten times, with different random seeds, and the mean values are presented. SkipNet uses 128-bit numeric IDs and a leaf set of 16 nodes. Chord and Pastry use their default configurations [27, 23].

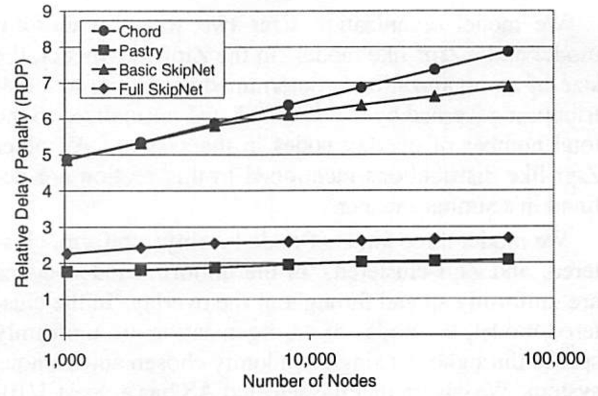


Figure 8. *RDP as a function of network size. Configuration: 1000 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

Chord	Basic SkipNet	Full SkipNet	Pastry
16.3	41.7	102.2	63.2

Table 1. *Average number of unique routing entries per node in an overlay with 2^{16} nodes.*

7.2 Basic Routing Costs

To understand SkipNet's routing performance we simulated overlay networks varying the number of nodes from 1,024 to 65,536. We ran experiments with 10, 100, and 1000 organizations and with all the permutations obtainable for organization size distribution, node placement, and data placement. The intent was to see how RDP behaves under various configurations. We were especially curious to see whether the non-uniform distribution of data object names would adversely affect the performance of SkipNet lookups, as compared to Chord and Pastry.

Figure 8 presents the RDPs measured for both implementations of SkipNet, as well as Chord and Pastry. Table 1 shows the average number of unique routing table entries per node in an overlay with 2^{16} nodes. All other configurations, including the completely uniform ones, exhibited similar results to those shown here.

Our conclusion is that basic SkipNet performs similarly to Chord and full SkipNet performs similarly to Pastry. This is not surprising since both basic SkipNet and Chord do not support network proximity-aware routing whereas full SkipNet and Pastry do. Since all our other configurations produced similar results, we conclude that SkipNet's performance is not adversely affected by non-uniform distributions of names.

7.3 Exploiting Locality of Placement

RDP only measures performance relative to IP-based routing. However, one of SkipNet's key benefits is that it enables localized placement of data. Figure 9 shows the average number of physical network hops for lookup requests. The x-axis indicates what fraction of lookups were forced to be to local data (i.e., the data object names

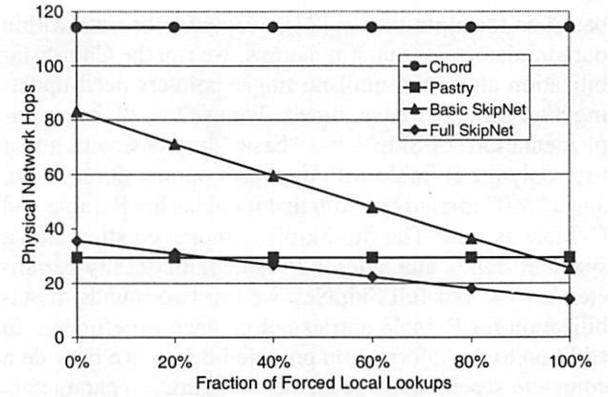


Figure 9. *Absolute latency (in network hops) for lookups as a function of data access locality (percentage of lookups forced to be within a single organization). Configuration: 2^{16} nodes, 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

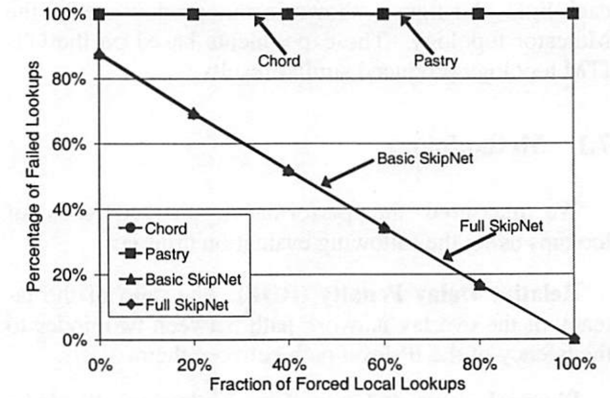


Figure 10. *Number of failed lookup requests as a function of data access locality (percentage of lookup requests forced to be within a single organization) for a disconnected organization. Configuration: 2^{16} nodes, 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

that were looked up were from the same organization as the requesting client). The y-axis shows the number of physical network hops for lookup requests.

As expected, both Chord and Pastry are oblivious to the locality of data references since they diffuse data throughout their overlay network. On the other hand, both versions of SkipNet show significant performance improvements as the locality of data references increases. It should be noted that Figure 9 actually understates the benefits gained by SkipNet because, in our Mercator topology, inter-domain links have the same cost as intra-domain links. In an equivalent experiment run on the GT-ITM topology, SkipNet end-to-end lookup latencies were over a factor of seven less than Pastry's for 100% local lookups.

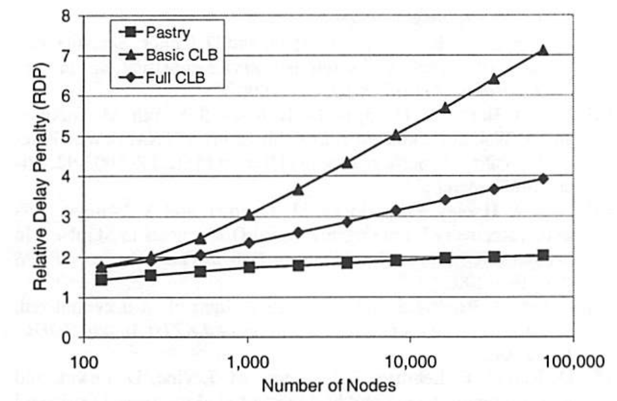


Figure 11. *RDP of lookups for data that is constrained load balanced (CLB) as a function of network size. Configuration: 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

7.4 Fault Tolerance

Content locality also improves fault tolerance. Figure 10 shows the number of lookups that failed when an organization was disconnected from the rest of the network.

This (common) Internet-like failure had catastrophic consequences for Chord and Pastry. The size of the isolated organization in this experiment was roughly 15% of the total nodes in the system. Consequently, Chord and Pastry will both place roughly 85% of the organization's data on nodes outside the organization. Furthermore, they must also attempt to route lookup requests with 85% of the overlay network's nodes effectively failed (from the disconnected organization's point-of-view). At this level of failures, routing is effectively impossible. The net result is a failed lookups ratio of very close to 100%.

In contrast, both versions of SkipNet do better the more locality of reference there is. When no lookups are forced to be local, SkipNet fails to access the 85% of data that is non-local to the organization. As the percentage of local lookups is increased to 100%, the percentage of failed lookups goes to 0.

7.5 Constrained Load Balancing

Figure 11 explores the routing performance of two different CLB configurations, and compares their performance with Pastry. For each system, all lookup traffic is organization-local data. The organization sizes as well as node and data placement are clustered with a Zipf-like distribution. The Basic CLB configuration uses only the R-Table described in Section 3, whereas Full CLB makes use of the R-Table and the C-Table, as described in Section 5.4.

The Full CLB curve shows a significant performance improvement over Basic CLB, justifying the cost of maintaining the extra routing tables. However, even with the additional tables, the Full CLB performance trails Pastry's performance. We plan to investigate further techniques to reduce the latency of CLB.

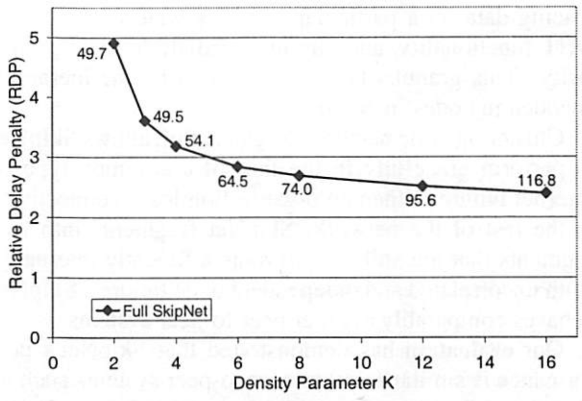


Figure 12. *RDP for Full SkipNet as a function of the density configuration parameter k. The labels next to each point represent the average number of unique pointers per node. Configuration:* 2^{16} *nodes, 1000 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

7.6 Network Proximity

Figure 12 shows the performance of SkipNet routing using the P-Table. The x-axis varies the configuration parameter k which controls the density of P-Table pointers. The y-axis shows the routing performance in terms of RDP, and each data point is labelled with the average number of unique pointers per node. Note that the C-Table was not enabled so the pointers are from the R-table, P-Table and leaf set. Figure 12 shows that for small values of k, increasing k yields a large RDP improvement with a small increase in the number of pointers. As k grows, we see minimal improvement in RDP but significantly more pointers. This suggests that choosing $k = 8$ provides most of the RDP benefit with a reasonable number of pointers.

We also analyzed the sensitivity of P-Table performance to the choice of the initial seed node. We compared the performance when choosing a seed node at random with choosing the seed as the closest node in the system. Our results show virtually identical performance, which indicates that the P-Table join mechanism is effective at locating a nearby seed.

8 Conclusion

To become broadly acceptable application infrastructure, peer-to-peer systems need to support both content and path locality: the ability to control where data is stored and to guarantee that routing paths remain local within an administrative domain whenever possible. These properties provide a number of advantages, including improved availability, performance, manageability, and security. To our knowledge, SkipNet is the first peer-to-peer system design that achieves both content and routing path locality. SkipNet achieves this without sacrificing the performance goals of previous peer-to-peer systems: Nodes maintain a logarithmic amount of state and operations require a logarithmic number of message hops.

SkipNet provides content locality at any desired degree of granularity. Constrained load balancing encompasses placing data on a particular node, as well as traditional DHT functionality, and any intermediate level of granularity. This granularity is only limited by the hierarchy encoded in nodes' name IDs.

Clustering node names by organization allows SkipNet to perform gracefully in the face of a common type of Internet failure: When an organization loses connectivity to the rest of the network, SkipNet fragments into two segments that are still able to route efficiently internally. With uncorrelated and independent node failures, SkipNet behaves comparably to other peer-to-peer systems.

Our evaluation has demonstrated that SkipNet's performance is similar to other peer-to-peer systems such as Chord and Pastry under uniform access patterns. Under access patterns where intra-organizational traffic predominates, SkipNet performs better. Our experiments show that SkipNet is significantly more resilient to organizational network partitions than other peer-to-peer systems.

In future work, we plan to deploy SkipNet across a testbed of 2000 machines emulating a WAN. This deployment should further our understanding of SkipNet's behavior in the face of dynamic node joins and departures, network congestion, and other real-world scenarios. We also plan to evaluate SkipNet as infrastructure for implementing a scalable event notification service [2].

Acknowledgements

We thank John Dunagan for his substantial help with both the analysis of SkipNet and with crafting the text of this paper. We thank Antony Rowstron, Miguel Castro, and Anne-Marie Kermarrec for allowing us to use their Pastry implementation and network simulator. We thank Atul Adya, who independently observed that Chord's structure suggested the possibility of a Skip List-based distributed data structure, and provided helpful feedback on drafts of this paper. Finally, we thank the anonymous conference referees for their insightful comments.

References

[1] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 2003.

[2] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *HotOS VIII*, May 2001.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for peer-to-peer routing overlays. In *Proceedings of the Fifth OSDI*, Dec. 2002.

[4] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[5] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS 2000*, pages 1–12, June 2000.

[6] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *18th Symposium on Operating Systems Principles*, Oct. 2001.

[8] J. R. Douceur. The Sybil Attack. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar. 2002.

[9] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd ICDCS*, July 2002.

[10] Gnutella. http://www.gnutelliums.com/.

[11] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth OSDI*, Oct. 2000.

[12] N. J. A. Harvey, J. Dunagan, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. Technical Report MSR-TR-2002-92, Microsoft Research, 2002.

[13] N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman. Efficient Recovery From Organizational Disconnects in SkipNet. In *Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[14] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proceedings of the 21st Annual PODC*, July 2002.

[15] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual STOC*, May 1997.

[16] P. Keleher, S. Bhattacharjee, and B. Silaghi. Are Virtualized Overlay Networks Too Much of a Good Thing? In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar. 2002.

[17] C. Labovitz and A. Ahuja. Experimental Study of Internet Stability and Wide-Area Backbone Failures. In *Fault-Tolerant Computing Symposium (FTCS)*, June 1999.

[18] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the 21st Annual PODC*, July 2002.

[19] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of 4th USITS*, Mar. 2003.

[20] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures*, 1989.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, Aug. 2001.

[22] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level Multicast using Content-Addressable Networks. In *Proceedings of the Third International Workshop on Networked Group Communication*, Nov. 2001.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[24] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th Symposium on Operating Systems Principles*, Oct. 2001.

[25] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communications*, Nov 2001.

[26] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, San Jose, CA, USA, Jan. 2002.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, Aug. 2001.

[28] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Proceedings of IEEE INFOCOM 2001*, April 2001.

[29] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd ICDCS*, July 2002.

[30] A. Vahdat, J. Chase, R. Braynard, D. Kostic, and A. Rodriguez. Self-Organizing Subsets: From Each According to His Abilities, To Each According to His Needs. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[31] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96*, April 1996.

[32] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, UC Berkeley, April 2001.

Symphony: Distributed Hashing In A Small World

Gurmeet Singh Manku
Stanford University
manku@cs.stanford.edu

Mayank Bawa
Stanford University
bawa@cs.stanford.edu

Prabhakar Raghavan
Verity Inc.
pragh@verity.com

Abstract

We present Symphony, a novel protocol for maintaining distributed hash tables in a wide area network. The key idea is to arrange all participants along a ring and equip them with long distance contacts *drawn from a family of harmonic distributions. Through simulation, we demonstrate that our construction is scalable, flexible, stable in the presence of frequent updates and offers small average latency with only a handful of long distance links per node. The cost of updates when hosts join and leave is small.*

1 Introduction

Peer to peer file sharing applications have surged in popularity in recent years. Systems like Napster, Gnutella, Kazaa and Freenet [4] have been used by millions of users. The research community is working on a wide variety of peer to peer applications like persistent data storage (CFS [6], Farsite [3], Oceanstore [11], PAST [19]), event notification and application level multicast (Bayeux [23], Scribe [20] and CAN-based Multicast [17]), DNS [5], resource discovery [2] and cooperative web caching [9]. Several of these applications have no centralized components and use a scalable distributed hash table (DHT) [8, 13, 16, 18, 22] as a substrate.

A DHT is a self-organizing overlay network of hosts that provides a service to add, delete and look up hash keys. Consider a network of n hosts over a wide area network that wish to cooperatively maintain a DHT with entries that change frequently. Replicating the entire hash table n times is unfeasible if n is large. One solution is to split the hash table into n blocks and let each host manage one block. This now requires a mapping table that maps a block id to its manager's network id. If n is of the order of hundreds or thousands of hosts and if hosts have short lifetimes, replicating this mapping table n times might be challenging. One possibility is to store this mapping table at a single central server, which would be consulted for every hash lookup. The load on the central server can be reduced by caching mapping table entries with DNS-style leases. However, a central server is a single point of failure and has to bear all traffic related to arrivals and departures.

Researchers have recently proposed distributed hashing protocols [8, 13, 16, 18, 22] that do not require any central servers. The mapping table is not stored explicitly anywhere. Instead, hosts configure themselves into a structured network such that mapping table lookups require a small number of hops. Designing a practical scheme along these lines is challenging because of the following desiderata:

Scalability: The protocol should work for a range of networks of arbitrary size.

Stability: The protocol should work for hosts with arbitrary arrival and departure times, typically with small lifetimes.

Performance: The protocol should provide low latency for hash lookups and low maintenance cost in the presence of frequent joins and leaves.

Flexibility: The protocol should impose few restrictions on the remainder of the system. It should allow for smooth trade-offs between performance and state management complexity.

Simplicity: The protocol should be easy to understand, code, debug and deploy.

We propose Symphony, a novel distributed hashing protocol that meets the criteria listed above. The core idea is to place all hosts along a ring and equip each node with a few *long distance links.* Symphony is inspired by Kleinberg's Small World construction [10]. We extend Kleinberg's result by showing that with $k = O(1)$ links per node, it is possible to route hash lookups with an average latency of $O(\frac{1}{k}\log^2 n)$ hops. Among the advantages Symphony offers over existing DHT protocols are the following:

Low State Maintenance: Symphony provides low average hash lookup latency with fewer TCP connections per node than other protocols. Low degree networks reduce the number of open connections and ambient traffic corresponding to pings, keep-alives and control information. Moreover, sets of nodes that participate in locking and coordination for distribute state update are smaller sized.

Fault Tolerance: Symphony requires f additional links per node to tolerate the failure of f nodes before a portion of the hash table is lost. Unlike other protocols, Symphony does not maintain backup links for each long distance contact.

Degree vs. Latency Tradeoff: Symphony provides a smooth tradeoff between the number of links per node and average lookup latency. It appears to be the only protocol that provides this tuning knob *even* at run-time. Symphony does not dictate that the number of links be identical for all nodes. Neither is the number stipulated to be a function of current network size nor is it fixed at the outset. We believe that these features of Symphony provides three benefits: (a) support for heterogeneous nodes, (b) incremental scalability, and (c) flexibility. We discuss these further in Section 5.3.

Road map: Section 2 explores related work. Section 3 describes Symphony. Section 4 contains experimental results. In Section 5, we compare Symphony with other protocols. Section 6 concludes the paper.

2 Related Work

2.1 Distributed Hash Tables

Plaxton, Rajaraman and Richa [15] devised a routing protocol based on hypercubes for a static collection of nodes. Routing is done by *digit-fixing*: e.g., when a node with id *zedstu* receives a query for *zedaaa*, it forwards it to a neighbor with prefix *zeda*. It turns out that for b bits per digit, each neighbor must maintain $O((2^b \log n)/b)$ neighbors resulting in $O((\log n)/b)$ worst case routing latency. Tapestry [8] adapted this scheme to a dynamic network for use in a global data storage system [11]. Pastry [18] is another scheme along the same lines where a node forwards a query to a neighbor with the longest matching prefix. In both Tapestry and Pastry, the number of bits per digit b is a configurable parameter that remains fixed at run-time.

CAN [16] embeds the key-space into a torus with d dimensions by splitting the key into d variable-length digits. A node forwards a query to the neighbor that takes it closer to the key. Nodes have $O(d)$ neighbors and routing latency is $O(dn^{1/d})$. The number of dimensions d is fixed in CAN. If the final network size can be estimated, then d could be made $O(\log n)$, resulting in $O(\log n)$ routing latency and $O(\log n)$ neighbors.

Chord [22] places participating nodes on a circle with unit perimeter. An m-bit hash key K is treated as a fraction $K/2^m$ for routing. Each node maintains connections with its immediate neighbors along the circle and a *finger table* of connections with nodes at distances approximately $\langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8} \ldots \rangle$ along the circle. Routing is done by forwarding to the node closest to, but not past, the key being looked up. Chord requires $O(\log n)$ neighbors and provides $O(\log n)$ routing latency. The protocols for joining and leaving the network introduce complexity and require $O(\log^2 n)$ messages each. A stabilization protocol is required to maintain network integrity.

Viceroy [13] is the first proposal that provides $O(\log n)$ routing latency with only a constant number of links. Like Chord, nodes are placed along a circle. A node additionally belongs to one out of approximately $O(\log n)$ concentric rings lying one above the other. These rings correspond to layers in Butterfly networks. A node maintains connections with two neighbors each along the two rings it belongs to. It also maintains two connections to a pair of nodes in a lower ring and one connection with a node in the ring above. Routing requires $O(\log n)$ hops on average.

DHT's over clusters have been extensively studied by the SDDS (Scalable Distributed Data Structures) community in the 90's. The term was coined by Litwin, Niemat and Shneider in their seminal paper [12]. Gribble et al. [7] implemented a highly scalable, fault tolerant and available SDDS on a cluster for Internet services. The requirements for a DHT over wide area networks are very different.

2.2 Small World Networks

Milgram conducted a celebrated experiment [14] that demonstrated the *Small World* phenomenon. He discovered that pairs of people in a society were connected by short chains of acquaintances. He also discovered that people were actually able to route letters to unknown persons in a few hops by forwarding them along a short path through acquaintanceships. To model the Small World phenomenon, Kleinberg [10] recently constructed a two-dimensional grid where every point maintains four links to each of its closest neighbors and just *one* long distance link to a node chosen from a suitable probability function. He showed that a message can be routed to any node by *greedy routing* in $O(\log^2 n)$ hops. Barriere et al. [1] studied Kleinberg's construction and proved its optimality under certain conditions.

Our work is inspired by Kleinberg's construction. We extend his result by showing that with $k = O(1)$ links, the routing latency diminishes to $O(\frac{1}{k} \log^2 n)$

hops. We also show how this basic idea can be adapted and engineered into a practical protocol for maintaining DHTs in a peer to peer network.

3 Symphony: The Protocol

Let I denote the unit interval $[0, 1)$ that wraps around. It is convenient to imagine I as a circle (ring) with unit perimeter. Whenever a node arrives, it chooses as its id a real number from I uniformly at random. A node manages that sub-range of I which corresponds to the segment on the circle between its own id and that of its immediate clockwise predecessor. A node maintains *two short links* with its immediate neighbors. Since all nodes choose their id's uniformly from I, we expect that they manage roughly equi-sized sub-ranges.

The nodes cooperatively maintain a distributed hash table. If a hash function maps an object to an m-bit hash key K, then the manager for this hash entry is the node whose sub-range contains the real number $K/2^m$. No restriction on m is imposed. Unlike CAN, Pastry and Tapestry, there is no relationship between m and the number or links.

3.1 Long Distance Links

Every node maintains $k \geq 1$ *long distance links.* For each such link, a node first draws a random number $x \in I$ from a probability distribution function that we will shortly define. Then it contacts the manager of the point x away from itself in the clockwise direction by following a *Routing Protocol* which we describe in Section 3.2. Finally, it attempts to establish a link with the manager of x.

We ensure that the number of incoming links per node is bounded by placing an upper limit of $2k$ incoming links per node. Once the limit is reached, all subsequent requests to establish a link with this node are rejected. The requesting node then makes another attempt by re-sampling from its probability distribution function. As a practical matter, an upper bound is placed on the number of such attempts, before a node gives up. We also ensure that a node does not establish multiple links with another node.

Probability distribution function (pdf): We denote the pdf by p_n, where n denotes the current number of nodes. The function $p_n(x)$ takes the value $1/(x \ln n)$ when x lies in the range $[1/n, 1]$, and is 0 otherwise. Drawing from p_n corresponds to a simple C expression: `exp (log(n) * (drand48() - 1.0))`, where `drand48()` produces a random number between 0 and 1. It is the continuous version of the discrete pdf proposed by Kleinberg. The distribution p_n belongs to a family of harmonic distributions. This observation inspired the name Symphony.

Estimation of n: Drawing from pdf p_n poses a problem: a node needs to know n to begin with. However, it is difficult for all nodes to agree on the exact value of current number of participants n, especially in the face of nodes that arrive and depart frequently. In Section 3.4, we will describe an *Estimation Protocol* that helps each of the nodes track n at run time. We denote an estimate of n by $\tilde{n}$. Thus a node draws from $p_{\tilde{n}}$ instead of p_n.

Choice of k: The number of links established by each node is a design parameter that is not fixed by our protocol. We experimentally show that as few as four long distance links are sufficient for low latency routing in large networks.

Long Distance Links in Practice: In a network with hosts spanning continents, we would have to embed the set of participating hosts onto a circle, taking network proximity into account. We expect this to require a fair amount of engineering and we are currently working on this problem. Once the circle has been established, we expect Symphony to be able to route lookups such that the latency does not exceed IP latency between the source and the final destination by a large factor. For example, CAN [16] demonstrated that a factor of two could be achieved for their construction for roughly $130K$ nodes.

The phrase "a network with k links per node" denotes a network where each node establishes 2 short links and k long links with other nodes. In terms of TCP connections, each node actually maintains an average of $t = 2k + 2$ connections.

3.2 Unidirectional Routing Protocol

When a node wishes to lookup a hash key $x \in I$, it needs to contact the manager of x.

> *A node forwards a lookup for x along that link (short or long) that minimizes the <u>clockwise distance</u> to x.*

Kleinberg [10] analyzed a static network in which each participant knows n precisely, has one long distance link corresponding to p_n, manages a sub-range of size $1/n$ and always routes clockwise. He showed that for $k = 1$ the expected path length for greedy routing is $O(\log^2 n)$ hops. We can show that in general, if each node has $k = O(1)$ links, expected path length is $O(\frac{1}{k} \log^2 n)$.

Theorem 3.1 *The expected path length with unidirectional routing in an n-node network with $k = O(1)$ links is $O(\frac{1}{k} \log^2 n)$ hops.*

Proof:

We sketch the proof assuming every attempted long-distance link is successful. However as noted above in Section 3.1, some of these connections in fact are rejected because the intended target of the link is "saturated" with $2k$ incoming links. We account for this implementation detail by noting that provided $k = O(1)$, the fraction of such rejected links is a constant and this only inflates the expected number of hops by a constant. We also assume that all nodes have accurate knowledge of n.

The pdf that we use for generating long distance neighbors is $p_n(x) = 1/(x \log n)$ for $x \in [1/n, 1]$ The probability p_{half} of drawing a value from $[z/2, z]$ for any $z \in [2/n, 1]$ is given by $\int_{z/2}^{z} p_n(x)dx = 1/\log_2 n$, which is independent of z. The significance of p_{half}: regardless of the current distance to the destination, it is the probability that any single long-distance link will cut the distance by at least half. The number of links to consider before the current distance diminishes by at least half follows a geometric distribution with mean $1/p_{\text{half}} = \log_2 n$. With k links per node, the expected number of nodes to consider before the current distance is at least halved is $\lceil (\log_2 n)/k \rceil$, which is less than $(2 \log_2 n)/k$ for $k \leq \log_2 n$.

Successive nodes along the route of a hash lookup diminish the distance to the destination. Consider that portion of the route where the distance is more than $2/n$. We showed that the expected number of nodes along the path before we encounter a node that more than halves the current distance is at most $(2 \log_2 n)/k$. The maximum number of times the original distance could possibly be halved before it is less than $2/n$ is $\log_2(n/2)$. Thus, the expected number of nodes along the route before the distance is less than $2/n$ is at most $2(\log_2 n)(\log_2(n/2))/k$. The remainder of the path consists of nodes that diminish the distance from $2/n$ to 0. The average contribution of these small path lengths is $O(1)$ because each node chose its id in the interval $[0, 1)$ uniformly at random. Thus the average path length of the full route is $O(\frac{1}{k} \log^2 n)$.

$\odot$

It is important that links be chosen following a harmonic distribution. Using the proof technique in [10], it can be shown that the average latency is $\Omega(\sqrt{n/k})$ if k links are chosen uniformly at random from the interval $[0, 1)$. Section 4.9 presents a graphical illustration of this observation.

3.3 Bidirectional Routing Protocol

In Section 3.1, we described how nodes establish long links with other nodes. The average number of *incoming links* is k per node. In practice, a link between two nodes would see continuous routing traffic. This would be materialized as a bidirectional TCP connection to leverage TCP's flow control, duplicate elimination and in-order delivery guarantees. If we were to use UDP, we would have to replicate much of this functionality.

One way to leverage incoming links is to treat them as additional long distance links. This helps reduce average latency only marginally. Much more benefit can be obtained by exploiting the following insight: The distribution of the source id of an incoming link corresponds roughly to p_n but in the anticlockwise direction. The observation that a node has exactly k clockwise and roughly k anticlockwise long distance links motivates the following protocol:

> *A node routes a lookup for x along that link (incoming or outgoing) that minimizes the <u>absolute distance</u> to x.*

Theorem 3.2 *The expected path length with bidirectional routing in an n-node network with $k = O(1)$ links is $O(\frac{1}{k} \log^2 n)$ hops.*

Proof: Along the same lines as Theorem 3.1. $\odot$

Bidirectional routing improves average latency by roughly 25% to 30% (See Section 4.2). Note that if we minimize absolute distance but restrict ourselves to clockwise movement, it is possible to get infinite loops.

With Bidirectional Routing, the average latency is still $O(\frac{1}{k} \log^2 n)$. However, the constant hidden behind the big-O notation is less than 1. We experimentally show that coupled with the idea of 1-Lookahead (see Section 3.7), networks as large as 2^{15} nodes have average latency no more than 7.5 for $k = 4$.

3.4 Estimation Protocol

Our Estimation Protocol is based on the following insight: Let X_s denote the sum of segment lengths managed by any set of s distinct nodes. Then $\frac{s}{X_s}$ is an unbiased estimator for n. The estimate improves as s increases. The idea of estimating n in this fashion is borrowed from Viceroy [13] where it was also shown that the estimate is asymptotically tight when $s = O(\log n)$. When the Estimation Protocol executes, all s nodes that contributed their segment lengths update their own estimates of n as well.

Choice of s: Our experiments show that $s = 3$ is good enough in practice. A node estimates n by using the length of the segment it partitions and its two neighboring segments. These three segment lengths are readily available at no extra cost from the two nodes between which x inserts itself in the ring. In Section 4.1, we show that the impact of increasing s on average latency is insignificant.

We note that an implementation of Symphony might employ a different Estimation Protocol. For example, an accurate value for n might be available from some central server where all participants must register before participating in the network. In another scenario, estimates of n could be piggybacked along with a small fraction of normal lookup messages, thereby amortizing the cost of maintaining $\tilde{n}$. A node might maintain the harmonic mean of the last few estimates it comes across. In this paper, we do not delve into sophisticated protocols for estimating n. For our purposes, a simple Estimation Protocol with $s = 3$ segments works fine.

3.5 Join and Leave Protocols

Join: To join the network, a new node must know at least one existing member. It then chooses its own id x from $[0, 1)$ uniformly at random. Using the Routing Protocol, it identifies node y, the current manager of x. It then runs the Estimation Protocol using $s = 3$, updating the estimates of three other nodes as well. Let $\tilde{n}_x$ denote the estimate of n thus produced. Node x then uses pdf $p_{\tilde{n}_x}$ to establish its long distance links. Since each link establishment requires a lookup that costs $O(\frac{1}{k}\log^2 n)$ messages, the total cost of k link establishments is $(\log^2 n)$ messages. The constant hidden behind the big-O notation is actually less than 1. See Section 4.6 for actual costs determined experimentally.

Leave: The departure of a node x is handled as follows. All outgoing and incoming links to its long distance neighbors are snapped. Other nodes whose outgoing links to x were just broken, reinstate those links with other nodes. The immediate neighbors of x establish short links between themselves to maintain the ring. Also, the successor of x initiates the Estimation Protocol over $s = 3$ neighbors, each of whom also updates its own estimate of n. The departure of a node requires an average of k incoming links to be re-established. The expected cost is $O(\log^2 n)$ messages. Again, the constant hidden behind the big-O notation is less than 1. See Section 4.6 for actual costs determined experimentally.

3.6 Re-linking Protocol

Each node x in the network maintains two values: $\tilde{n}_x$, its current estimate of n and $\tilde{n}_x^{link}$, the estimate at which its long distance links were last established. Over its lifetime, $\tilde{n}_x$ gets updated due to the Estimation Protocol being initiated by other nodes. Whenever $\tilde{n}_x \neq \tilde{n}_x^{link}$, it is true that the current long distance links of x correspond to a stale estimate of n. One solution is to establish all links afresh. However, if a node were to re-link on *every* update of $\tilde{n}_x$, traffic for re-linking would be excessive. This is because re-establishment of all k long distance links requires $O(\log^2 n)$ messages.

Re-linking Criterion: A compromise re-linking criterion that works very well is to re-link only when the ratio $\tilde{n}_x/\tilde{n}_x^{link} \notin [1/2, 2]$. The advantage of this scheme is that as n steadily grows or shrinks, traffic for re-linking is smooth over the lifetime of the network. In particular, if nodes arrive sequentially and each node knows n precisely at all times, then the number of nodes re-linking at any time would be at most one. We experimentally show that even in the presence of imprecise knowledge of n, the re-linking cost is smooth over the lifetime of a network. However, we also show that the benefits of re-linking are marginal.

3.7 Lookahead

Two nodes connected by a long link could periodically exchange some information piggy-backed on keep-alives. In particular, they could inform each other about the positions of their respective long distance contacts on the circle. Thus a node can learn and maintain a list of all its neighbor's neighbors. We call this the lookahead list. The lookahead list helps to improve the choice of neighbor for routing queries. Let u denote the node in the list that takes a query closest to its final destination. Then the query is routed to that neighbor that contains u in its neighbor set. Note that we do not route directly to u. The choice of neighbor is not greedy anymore. If hash lookups are exported by an RPC-style interface, (e.g., iterative/non-recursive queries in DNS), we *could* forward a client to a neighbor's neighbor, cutting down average latency by half. Upon receiving a forwarded lookup request, a neighbor makes a fresh choice for its own best neighbor to route to.

We experimentally show that 1-Lookahead effectively reduces average latency by roughly 40%.

What is the cost of 1-Lookahead? The size of the lookahead list is $O(k^2)$. The number of long links remains unchanged because a node does not

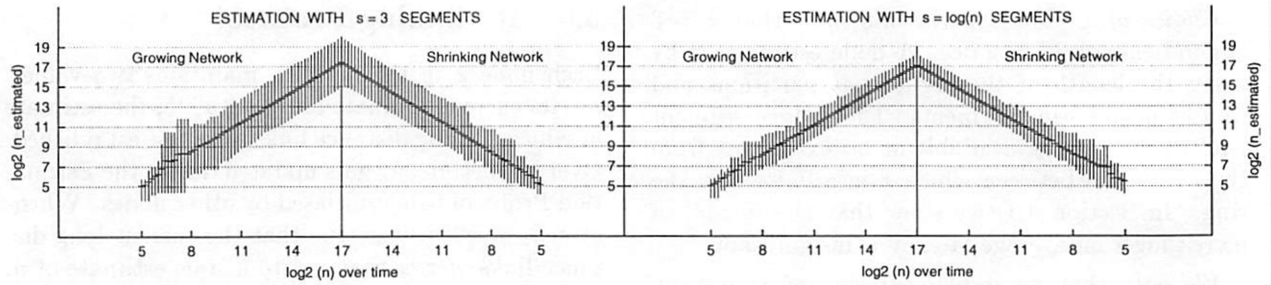


Figure 1: *Quality of estimated value of n as the network first expands and then shrinks. Each vertical line segment plots the average along with an interval that captures 99% of the distribution.*

directly link to its neighbors' neighbors; it just remembers their id's. However, arrival and departure of any node requires an average of $k(2k+2)$ messages to update lookahead lists at $k(2k+2)$ nodes in the immediate neighborhood. These messages need not be sent immediately upon node arrival/departure. They are sent lazily, piggy-backed on normal routing packets or keep-alives exchanged between pairs of nodes. Lazy update of lookahead lists might introduce temporary inconsistencies. This is acceptable because routing does not crucially depend on these lists. Lookaheads just provide a better hint. We are currently investigating further the role of approximate lookahead.

We could employ ℓ-Lookahead in general, for $\ell > 1$. However, the cost of even 2-Lookahead becomes significant since each update to a link would now require $O(k^3)$ additional messages for updating lookahead lists. If ℓ were as large as $O(\log^2 n)$, each node could effectively compute the shortest path to any destination.

4 Experiments

In this section, we present results from our simulation of Symphony on networks ranging from 2^5 to 2^{15} nodes in size. We systematically show the interplay of various variables (n, k and s), justifying our choices.

We study four kinds of networks: A STATIC network with n nodes is constructed by placing n nodes on a circle, splitting it evenly into n segments. Knowledge of n is global and accurate. An EXPANDING network is one that is constructed by adding nodes to the network sequentially. An estimate of n is used to establish long distance links. An EXPANDING-RELINK network is simply an EXPANDING network in which nodes re-establish links using the re-linking criterion mentioned in Section 3.6. Finally, a DYNAMIC network is one in which nodes not

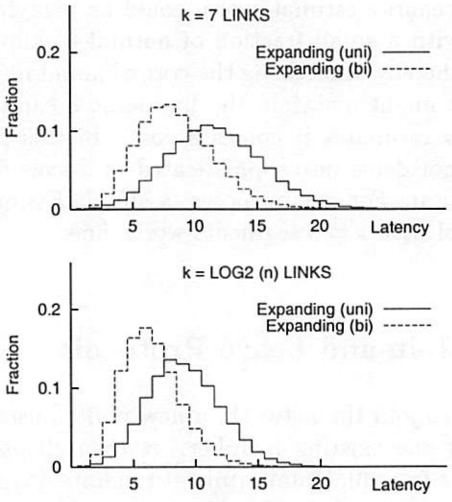


Figure 2: *Latency distributions for a network with 2^{14} nodes.*

only arrive but also depart. We describe the exact arrival and departure distributions in Section 4.4.

4.1 Estimation Protocol

Figure 1 shows performance of the Estimation Protocol when a network grew from zero to 2^{17} nodes and then shrank. Each vertical segment in the figure captures 99% of the nodes. The Estimation Protocol tracks n fairly well. The estimate is significantly improved if we use $s = \log \tilde{n}$ neighbors, where $\tilde{n}$ itself is obtained from any existing node. However, the impact on average latency is not significant, as we show in Section 4.3. All our experiments described hereafter were conducted with $s = 3$.

4.2 Routing Protocol

Figure 3 plots the average latency for three networks: STATIC, EXPANDING and EXPANDING-RELINK. The

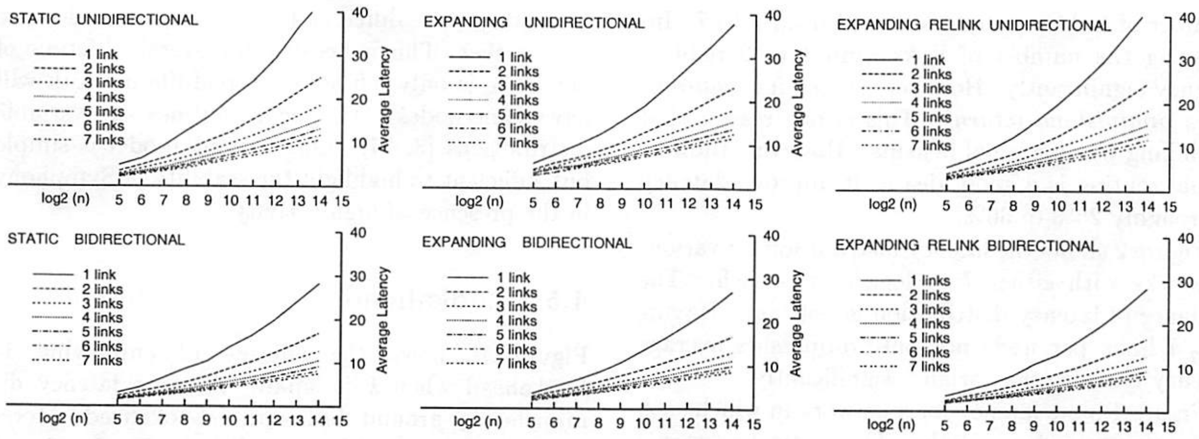


Figure 3: *Average latency for various numbers of long distance links and n ranging from* 2^5 *and* 2^{14}.

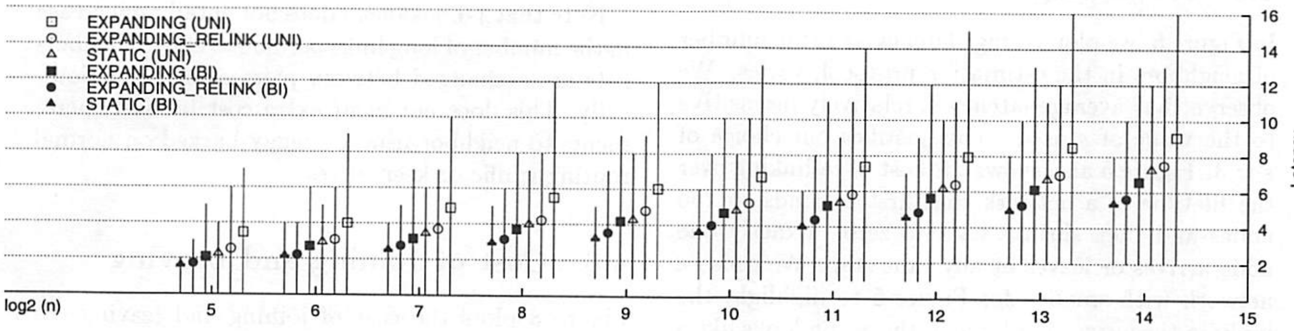


Figure 4: *Latency for various networks with* $\log_2 \tilde{n}$ *links per node. Each vertical segment plots the average along with an interval that captures 99% of the distribution.*

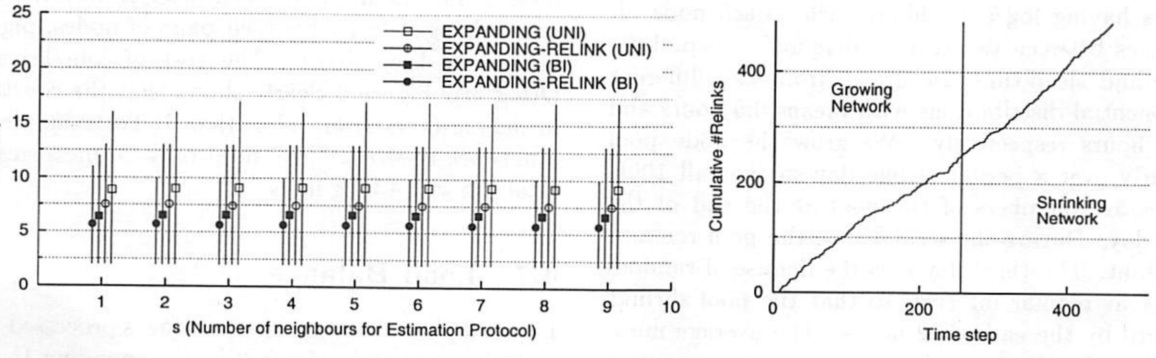


Figure 5: **Left:** *Latency for* EXPANDING *networks using Estimation Protocol with various values of s, the number of neighbors contacted for estimating n.* **Right**:*Cumulative number of re links for a network that first expands from 0 to 256 nodes and then shrinks back to 0. At every time step, exactly one node joins or leaves.*

number of links per node is varied from 1 to 7. Increasing the number of links from 1 to 2 reduces latency significantly. However, *successive additions have diminishing returns.* This is one reason that re-linking has marginal benefits. However, Bidirectional routing is a good idea as it improves latency by roughly 25% to 30%.

Figure 2 shows the latency distribution for various networks with either 7 or $\log_2 \tilde{n}$ links each. The variance of latency distribution is not high. Having $\log_2 \tilde{n}$ links per node not only diminishes average latency but also the variance significantly.

Figure 4 plots latency for a network in which each node maintains $\log_2 \tilde{n}$ links. The vertical segments capture 99% of node-pairs. For a given type of network, average latency grows linearly with $\log n$, as expected.

4.3 Re-linking Protocol

In Figure 5, we plot average latency as s, the number of neighbors in the estimation protocol, varies. We observe that average latency is relatively insensitive to the value of s used. This justifies our choice of $s = 3$. Figure 5 also shows the cost of re-linking over the lifetime of a network that first expands to 256 nodes and then shrinks back to zero. Exactly one node arrives or leaves at any time step. We chose a network with small n for Figure 5 to highlight the kinks in the curve. For large n, the graph looks like a straight line. The cost of re-linking is fairly smooth.

4.4 Dynamic Network

A DYNAMIC network is one in which nodes arrive and depart. We studied a network with n = 100K nodes having $\log \tilde{n}$ neighbors each. Each node alternates between two states: alive and asleep. Lifetime and sleep-time are drawn from two different exponential distributions with means 0.5 hours and 23.5 hours respectively. We grow the node pool linearly over a period of one day so that all 100K nodes are members of the pool at the end of the first day. During the second day, the pool remains constant. The third day sees the demise of random nodes at regular intervals so that the pool shrinks to zero by the end of 72 hours. The average number of nodes that participate in the ring at any time is $\frac{0.5}{0.5+23.5} \times 100K \approx 2K$. From Figure 6, we see that the Estimation Protocol is able to track n sufficiently accurately and that the average latency was always less than 5 hops.

We wish to point out that the network we simulated is very dynamic. The set of nodes at any point of time is quite different from the set of nodes one hour earlier. This is because the average lifetime of any node is only 0.5 hour. A real-life network will have some nodes with longer lifetimes and variable distributions [3, 21]. Our current model is simple but sufficient to highlight the stability of Symphony in the presence of high activity.

4.5 Lookahead

Figure 7 shows the efficacy of employing 1-Lookahead when k is small. Average latency diminishes by around 40% with 1-Lookahead. Moreover, the spread of latency distribution (captured by vertical line segments in the graph) shrinks. For a network with 2^{15} nodes, average latency is 7.6 with $k = 4$. We also simulated 1-Lookahead with $k = \log \tilde{n}$ links per node and saw average latency drop to 4.4.

Note that 1-Lookahead does not entail an increase in the number of long links per node. Only neighbor-lists are exchanged between pairs of nodes periodically. This does not incur extra cost because increments to neighbor-lists are piggy-backed on normal routing traffic or keep-alives.

4.6 Cost of Joining and Leaving

Figure 8 plots the cost of joining and leaving the network. Whenever a node joins/leaves, k long distance links have to be created apart from updates to short links. Join/leave cost is proportional to the number of links to be established. The cost diminishes as average lookup latency drops. When using 1-Lookahead, there are an additional $k(2k + 2)$ messages to update lookahead lists. However, these are exchanged lazily between pairs of nodes, piggy-backed on keep-alives. The cost of join/leave is $O(\log^2 n)$. Figure 8 clearly shows that the constant in the big-O notation is less than 1. For example, in a network of size 2^{14}, we need only 20 messages to establish $k = 4$ long links.

4.7 Load Balance

Figure 9 plots the number of messages processed per node in a network of size 2^{15} corresponding to 2^{15} lookups. Each lookup starts at a node chosen uniformly at random. The hash key being looked up is also drawn uniformly from $[0, 1]$. The routing load on various nodes is relatively well balanced. Both the average and the variance drop when we employ 1-Lookahead. Curiously, the distribution is bimodal

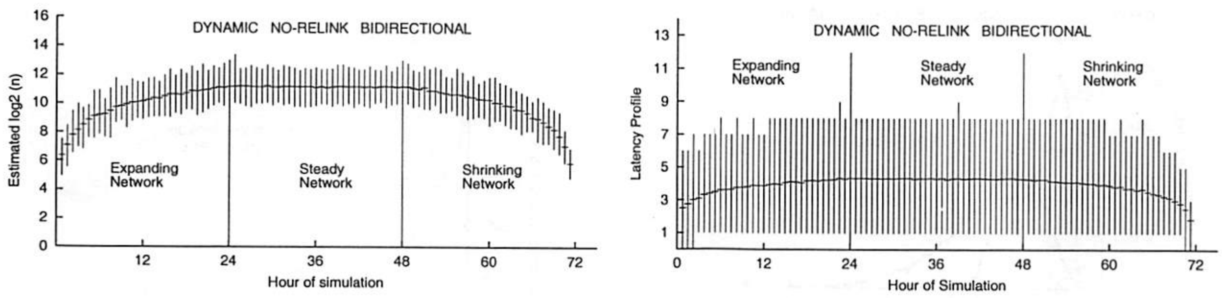


Figure 6: *Performance of a* DYNAMIC *network of 100K nodes with* $\log \tilde{n}$*-links using the Estimation Protocol but no re-linking. Each node is alive and asleep on average for 0.5 hours and 23.5 hours respectively. The node pool linearly increases to 100K over the first day. The pool is steady on the second day. A random node departs at regular intervals on the third day until the network shrinks to zero. Each vertical segment plots the average along with the range of values that covers 99% of the distribution.*

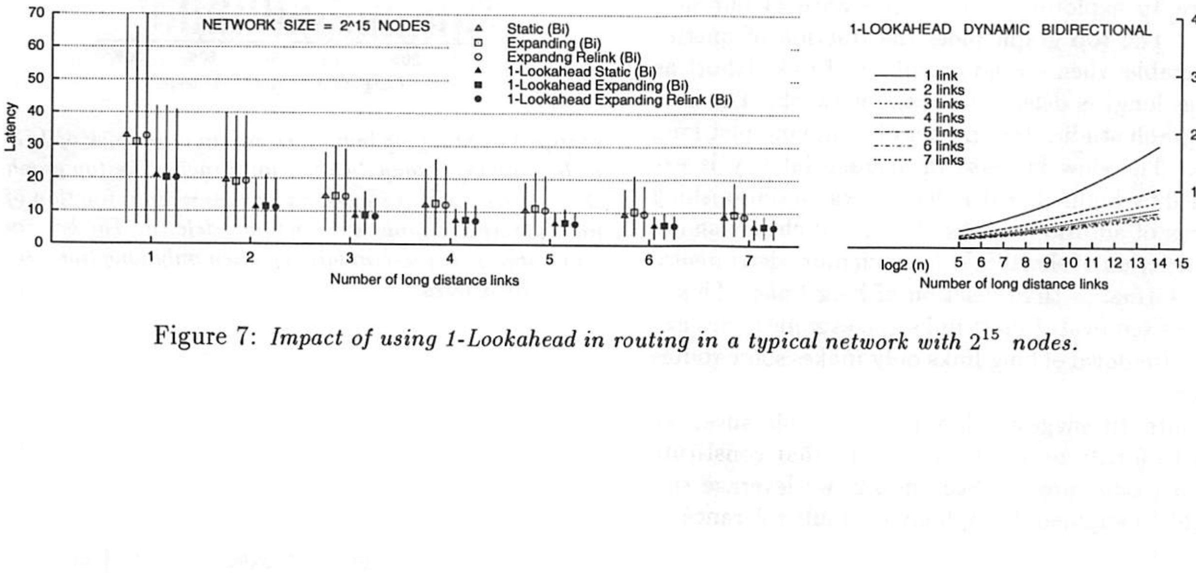


Figure 7: *Impact of using 1-Lookahead in routing in a typical network with* 2^{15} *nodes.*

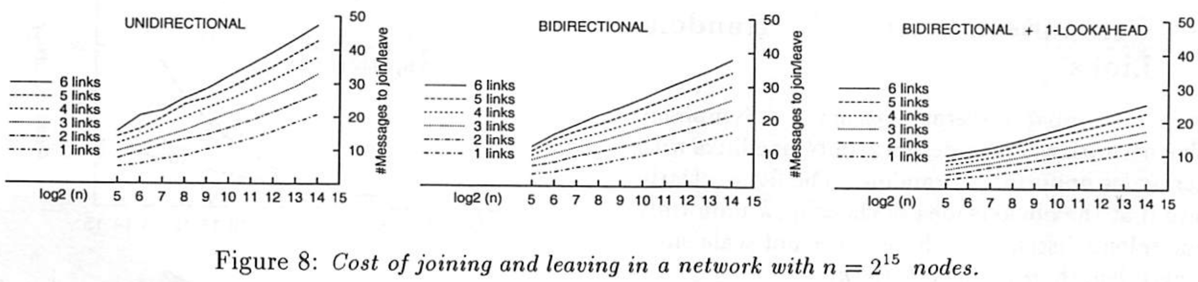


Figure 8: *Cost of joining and leaving in a network with* $n = 2^{15}$ *nodes.*

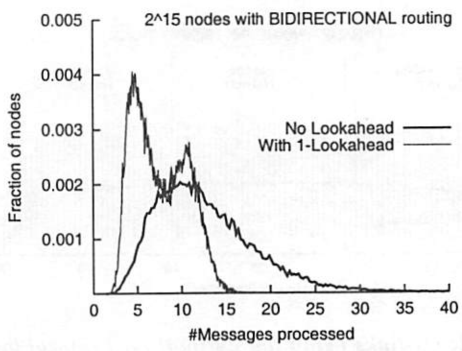


Figure 9: *Bandwidth profile in a network with $n = 2^{15}$ nodes with $k = 4$ links per node. Each node looks up one random hash key.*

when 1-Lookahead is employed. We are investigating the reason for this behavior.

4.8 Resilience to Link Failures

Figure 10 explores the fault tolerance of our network. The top graph plots the fraction of queries answerable when a random subset of links (short as well as long) is deleted from the network. The bottom graph studies the impact of removing just long links. The slow increase in average latency is explained by Figure 3 which demonstrated diminishing returns of additional links. Figure 10 clearly shows that deletion of short links is much more detrimental to performance than deletion of long links. This is because removal of short links makes some nodes isolated. Removal of long links only makes some routes longer.

Figure 10 suggests that for fault tolerance, we need to fortify only the short links that constitute the ring structure. In Section 5.2, we leverage this insight to augment Symphony for fault tolerance.

4.9 Comparison with k Random Links

Figure 11 compares average latency for Symphony with a network where nodes form outgoing links with other nodes uniformly at random. The figure clearly shows that the obvious idea of choosing k uniformly random long distance neighbors does not scale since the path length grows as $O(\sqrt{n/k})$.

5 Comparison and Analysis

Symphony is a simple protocol that scales well and offers low lookup latency with only a handful of TCP

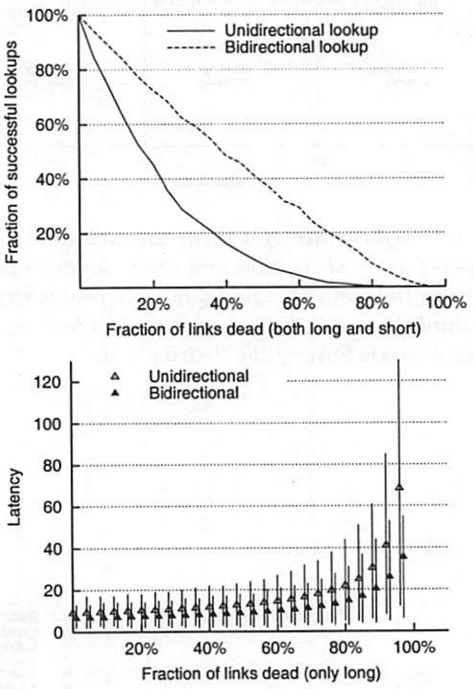


Figure 10: *Studying fault tolerance in a network of 16K nodes with $\log \tilde{n}$ long distance links each. The top graph shows percentage of successful lookups when a fraction of links (short and long) are randomly deleted. The bottom graph shows increase in latency when only long links are randomly deleted.*

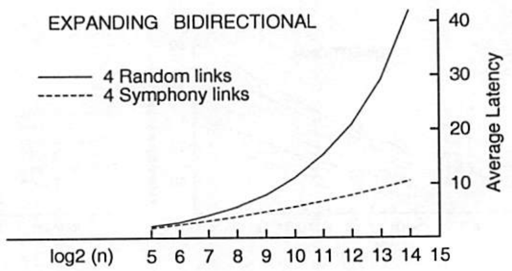


Figure 11: *Comparison with a network where each node links to k other nodes chosen uniformly at random. Network size $n = 2^{15}$ nodes.*

connections per node. The cost of joining and leaving the network is small. Symphony is stable in the presence of frequent node arrivals and departures. We now highlight features unique to Symphony.

5.1 Low State Maintenance

Table 1 lists lookup latency vs. degree for various DHT protocols. Low degree networks are desirable for several reasons. First, fewer links in the network reduce the average number of open connections at servers and reduce ambient traffic corresponding to pings/keep-alives and control information. Second, arrivals and departures engender changes in DHT topology. Such changes are concomitant with the state update of a set of nodes whose size is typically proportional to the average degree of the network. Fewer links per node translates to smaller sets of nodes that hold locks and participate in some coordination protocol for distributed state update. Third, small out-degree translates to smaller bootstrapping time when a node joins and smaller recovery time when a node leaves without notice. Finally, it should be easier to isolate faults in low degree networks, making debugging faster. Symphony actually provides a smooth tradeoff between average latency and the amount of state per node.

5.2 Fault Tolerance

Symphony allows replication of content for fault tolerance by making f copies of a node's content at each of the f nodes succeeding it in the clockwise direction. A node maintains direct connections with all its f successors. This arrangement can tolerate f simultaneous failures before a portion of the hash table is lost. When a new key-value pair is inserted, a node propagates the changes to its f successors for consistency. Furthermore, all lookups succeed until some f successive nodes fail together. This is because the network remains connected (the base circle is intact) as long as at least one out of f successive nodes is alive for each node. Overall, the average number of TCP connections per node is $2k + 2 + f$. In practice, a small value of f less than ten should suffice, assuming independent failures and short recovery times.

Symphony's design for fault tolerance was motivated by Figure 10 (Section 4.8) where we showed that it is the short links that are crucial for maintaining connectivity. CFS [6] and PAST [19] use a variant of our fault tolerance scheme.

Symphony does not create any redundant long distance links for fault tolerance. There are no *backup* long links. It is only the short links that are fortified by maintaining connections with f successors per node. The long links contribute to the *efficiency* of the network; they are not critical for correctness (see Section 4.8). Protocols like Pastry, Tapestry and CAN maintain two to four backup links for *every* link a node has. A glance at Table 1 reveals that the overhead of redundant links for fault tolerance is significantly less for Symphony than other protocols. Having fewer links per node has several other benefits that we described in the preceding section.

5.3 Degree vs. Latency Tradeoff

Symphony provides a smooth tradeoff between the number of links per node and average lookup latency. It appears to be the only protocol that provides this tuning knob *even* at run-time. Symphony does not dictate that the number of links be identical for all nodes. Neither is the number stipulated to be a function of current network size nor is it fixed at the outset. We believe that these features of Symphony provides three benefits:

Support for Heterogeneous Nodes: Each node is merely required to have a bare minimum of two short-distance links. The number of long-distance links can be chosen for each individual node according to its available bandwidth, average lifetime, or processing capability. All the other DHT protocols specify the exact number and identity of neighbors for each node in the network. It is not clear how they would accommodate nodes with variable degrees. Symphony's randomized construction makes it adapt naturally to heterogeneous nodes ranging from home computers with dial-in connections to LAN-based office computers.

Incremental Scalability: Symphony scales gracefully with network size. The Estimation Protocol provides each participant with a reasonably accurate estimate of network size. It is possible for nodes to adapt the number of long distance links in response to changes in network size to guarantee small average lookup latency. This obviates the need to estimate in advance the maximum size of the network over its lifetime.

Flexibility: An application designer who uses a distributed hash table (DHT) would want to make its implementation more efficient by leveraging knowledge unique to the problem scenario. For example, the specifics of the network topology at hand, or the behavior of participating hosts, or a priori knowledge about the load on the DHT might be known. If the DHT itself has a rigid structure, the application designer is severely constrained. Sym-

# TCP Connections	Lookup Latency	Protocol	# TCP Connections	Lookup Latency	Notes
$2d$	$(d/2)n^{\frac{1}{d}}$	CAN	20	14.14	Fixed #dimensions
$2\log_2 n$	$(\log_2 n)/2$	Chord	30	7.50	Fixed #links
10	$\log_2 n$	Viceroy	10	15.00	Fixed #links
$(2^b-1)(\log_2 n)/b$	$(\log_2 n)/b$	Tapestry	56	3.75	with b=4 digits
$(2^b-1)(\log_2 n)/b$	$(\log_2 n)/b$	Pastry	22	7.50	with b=2 digits
			56	3.75	with b=4 digits
$2k+2$	$c(\log^2 n)/k$	Symphony	10	7.56	k=4, bidirectional with 1-lookahead
			56	3.75	k=27, bidirectional with 1-lookahead

Table 1: *Comparison of various protocols for a network of size 2^{15}. Latencies are measured in terms of hops along the respective network topologies.*

phony allows the number of links to be variable. All outgoing links are *identical* in the sense that they are drawn from the same probability distribution function. We believe that the randomized nature of Symphony poses few constraints as compared with other protocols.

5.4 Comparison with Other Protocols

We compare Symphony with other DHT protocols for a network of size $n = 2^{15}$ nodes. We also discuss how other protocols could possibly use 1-lookahead and deploy additional links, if available.

(a) CAN can route among n nodes with an average latency of $(d/2)n^{\frac{1}{d}}$. The optimal value of d for $n = 2^{15}$ nodes is 10 resulting in an average latency of 14.14. The average number of TCP connections is $2d = 20$. Dimensionality in CAN is fixed at the outset. It is not clear how dimensionality can be dynamically changed as the network expands or shrinks. Thus, CAN nodes would have 20 TCP connections each even if the network size is small. Unlike other protocols, CAN runs a *zone rebuilding protocol* in the background to adjust hash table partitions. CAN has low cost of joining. Heuristics for constructing a CAN topology that is aware of real network proximity have been shown to yield low IP latency on synthetic networks [16].

(b) Chord stipulates that every node in a network with 2^{15} must have $\log_2 n = 15$ outgoing links each, with the result that average latency is 7.5. In terms of TCP connections, nodes have $2\log_2 n = 30$ connections each. Among existing DHT protocols, Symphony is closest in spirit to Chord. Chord could borrow ideas from Symphony for better performance. For example, Chord currently uses clockwise routing using unidirectional links. It can be modified to employ Symphony-style greedy routing over bidirectional links that minimizes absolute distance to the target at each hop. Chord uses a rather expensive re-linking and stabilization protocol upon every insertion and deletion. When a node joins, up to $O(\log n)$ other nodes who were pointing to this node's successor might have to re-establish their link with the new node. Experience with Symphony shows that re-linking is not worthwhile and that greedy routing continues to work satisfactorily even when nodes do not re-link.

(c) Pastry, with a digit size of 2 bits, would need an average of 22 TCP connections per node for average latency 7.5. Pastry can improve the latency to 3.75, but only with as many as 56 TCP connections per node. The digit size is a parameter that is fixed at the outset. For fault tolerance, Pastry maintains backup links for every link in its routing table. Moreover, content is replicated among L adjacent nodes. Pastry exploits network locality while choosing id's of nodes. The average latency over synthetic networks has been shown to be small [18].

(d) Tapestry uses 4-bit digits resulting in average lookup latency of 3.75 with 56 links per node. Tapestry is very similar to Pastry. The digit size is a parameter that is fixed at the outset. For fault tolerance, Pastry maintains backup links for every link in its routing table.

(e) Viceroy maintains seven links per node, irrespective of n, the size of the network. Each node has two neighbors along two rings, one up-link and two down-links. Four of these links are bidirectional, three are unidirectional. Thus, a Viceroy node would actually have an average of $t = 10$ TCP connections per node. For $n = 2^{15}$, the average latency in Viceroy would be at least $\log(n) = 15$. This corresponds to an average 7.5 levels to reach *up* to the highest ring and another 7.5 levels to come *down* to the ring at the right level. Viceroy appears to be more complex and it is not clear how it would exploit network proximity while maintaining multiple concentric rings.

(f) Symphony offers a wide variety of choices for the number of TCP connections for a fixed value of $n = 2^{15}$ nodes. Figure 7 shows that the average latency with $k = 4$ long links with 1-Lookahead and bidirectional routing is 7.6. Such a topology results in 10 TCP connections per node on average. As k increases, Symphony's average latency reduces. Symphony does not use backup links for long distance links. Instead each node replicates content on f successor nodes and maintains direct connections with them. This ensures content availability and successful lookups as long as no string of f successive nodes fails.

Lookahead seems to be of little value to CAN, Pastry and Tapestry. This is because the route of a lookup (and therefore, its length) is fixed. The protocol would not choose a different route if 1-Lookahead lists were available. The reason why lookahead is useful in Symphony is that a message could follow several paths from a source to a destination. We suspect that Chord might benefit by employing 1-Lookahead.

Let us see how additional links could possibly be used by various DHT protocols to improve performance. Chord could use a larger finger table. To change average degree at run-time in response to changes in network sizes, Chord could deploy Symphony's Estimation protocol for estimating n. A CAN node maintains connections with its immediate neighbors in a d-dimensional torus. A natural way for CAN to use more links would be to increase its dimensionality. However, this is a fixed parameter and it is not clear how dimensionality could be changed at run time cleanly. CAN could presumably use additional links to connect to farther nodes along each dimension, giving it a flavor of Chord/Symphony per dimension. Viceroy uses seven links no matter how large n is. It remains unclear how Viceroy would employ more links if available; presumably it would entail a construction built on a d-way butterfly for $d > 2$. Pastry and Tapestry could use more links by increasing the digit-size which is a parameter fixed at the outset. Since the expected number of links is given by the formula $(2^b \log n)/b$, the possible values of average out-degree are limited and far apart. Symphony and a modified version of Chord appear to be the only protocols that offer a smooth trade-off between average latency and number of links per node.

6 Conclusions

We have presented Symphony, a simple protocol for managing a distributed hash table in a dynamic network of hosts with relatively short lifetimes. Through a series of systematic experiments, we have shown that Symphony scales well, has low lookup latency and maintenance cost with only a few neighbors per node. In particular, $s = 3$ neighbors suffice for the Estimation Protocol and $k = 4$ long distance links with Bidirectional routing and 1-Lookahead are sufficient for low latencies in networks as big as 2^{15} nodes. We believe that Symphony is a viable alternative to existing proposals for distributed hashing.

We plan to adapt Symphony to an environment with heterogeneous nodes and gain experience with the implementation we are currently working on. An important next step in implementation is to take network proximity between nodes and heterogeneity into account.

7 Acknowledgments

We thank Geoff Voelker for his insightful comments that went a long way in improving the paper. We also thank the anonymous referees, Shankar Ponnekanti and Ramesh Chandra for their feedback. This work was partially supported by a grant from SNRC.

References

[1] L. Barriere, P. Fraigniaud, E. Kranakis, and D. Krizanc. Efficient routing in networks with long range contacts. In *Proc. 15th Intl. Symp. on Distributed Computing (DISC 01)*, pages 270–284, 2001.

[2] M. Bawa, G. S. Manku, and P. Raghavan. SETS: Search Enhanced by Topic Segmentation. *Submitted for publication*, 2003.

[3] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS 2000*, pages 34–43, 2000.

[4] I. Clarke, T. Hong, S. Miller, O. Sandberg, and B. Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[5] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving dns using a peer-to-peer lookup service. In *Proc. 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS 2002)*, pages 155–165, 2002.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 202–215, 2001.

[7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 319–332, 2000.

[8] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 41–52, 2002.

[9] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 213–222, 2002.

[10] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC 2000)*, pages 163–170, 2000.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. 9th Intl. conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, 2000.

[12] W. Litwin, M. Neimat, and D. A. Schneider. Lh*-a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[13] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 183–192, 2002.

[14] S. Milgram. The small world problem. *Psychology Today*, 67(1), 1967.

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA 1997)*, pages 311–320, 1997.

[16] S. Ratnasamy, P. Francis, M. Handley, and R. M. Karp. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001*, pages 161–172, 2001.

[17] S. Ratnasamy, S. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content addressable networks. In *Proc. 3rd Intl. Networked Group Communication Workshop (NGC 2001)*, pages 14–29, 2001.

[18] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, 2001.

[19] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 188–201, 2001.

[20] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. 3rd Intl. Networked Group Communication Workshop (NGC 2001)*, pages 30–43, 2001.

[21] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN'02)*, 2002.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, pages 149–160, 2001.

[23] S. Zhang, B. Zhao, B. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for wide-area, fault-tolerant data dissemination. In *Proc. 11th Intl. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, pages 11–20, 2001.

A Study of the Performance Potential of DHT-based Overlays

Sushant Jain Ratul Mahajan David Wetherall

{sushjain,ratul,djw}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

Abstract

We use simulation to study whether overlays based on the recent distributed hash tables (DHTs) have the potential to deliver performance comparable to that of overlays based on measurements. Our work is motivated by the use of DHTs for services such as multicast, which is already targeted by measurement-based overlays; there is currently little understanding of how the two approaches compare at scales where both are viable.

We compare three DHT-based overlays (CAN, Chord and Pastry) with two measurement-based overlays (Narada and NICE), as well as power-law random graphs (PLRGs) that represent Gnutella. To enable comparisons, we configure the overlays with the same average out-degree and focus on moderate scale. To gauge potential, we look at current and idealized DHT algorithms. We find that basic versions of DHTs have a latency stretch that is at least twice that of NICE and Narada, but similar performance in terms of bandwidth hotspots. However, DHT performance can be improved considerably with routing heuristics and topology-aware overlay construction, which have the potential to bring DHT performance at par with NICE. We also report on performance of overlays with power-law structure and the impact of hierarchy on performance.

1 Introduction

Overlays have become the preferred vehicle for providing new Internet services, e.g., CDNs [1, 12], application-level multicast [15, 4, 17, 16, 20, 25, 7, 39, 20, 34], and P2P file sharing [18, 13, 10, 22, 11]. As such, overlay construction protocols that provide good levels of performance, scalability, and robustness are of considerable importance. There has been a surge of interest in the area, along with rapid advances that have led to two main approaches to overlay construction: protocols based on measurements, and protocols based on the recently developed distributed hash tables (DHTs).

Measurement-based protocols use estimates of network properties such as latency between overlay nodes to make an informed choice of overlay structure. They were initially motivated by application-level multicast. Examples include Narada [15], RON [2], NICE [4], Kudos [16] and TAG [20]. These algorithms provide good performance via high quality paths. Recent extensions have allowed them to scale up to tens of thousands of nodes by using hierarchy [4, 19].

On the other hand, DHT-based protocols begin with structure in mind, and may fold measurement information into that structure. They map the overlay nodes to a virtual space commonly known as the node identifier space, typically pseudo-randomly to achieve a balanced distribution. The overlay topology, which determines how the nodes connect to each other, is governed mainly by these node identifiers. Examples include CAN [27], Chord [30], Pastry [28] and Tapestry [37]. These algorithms are extremely scalable and were motivated by peer-to-peer (P2P) applications such as distributed file sharing, where millions of nodes may be involved.

While originally developed for different purposes, DHT-based overlays are now being targeted at some of the same applications as measurement-based overlays, most visibly application-level multicast [25, 7, 31, 39]. Here, efficient use of the network is a key concern, more so than for earlier DHT applications such as distributed indexing. Much recent work thus aims to better the performance of DHT-based overlays with improved construction heuristics [5, 36, 38, 24, 32]. However, despite this convergence of purpose and plenitude of work, there is no real understanding of how the two approaches compare.

In this paper, we study the performance of DHT-based overlays at moderate scales (1000s of nodes) where they represent an alternative to measurement-based overlays for multicast and other services. In contrast, most other work on DHTs studies scales up to hundreds of thousands or millions of nodes. We seek to determine whether DHT-based overlays have the potential to deliver performance that is comparable to measurement-based overlays at our scales. If so, then research on improved heuristics is more likely to be fruitful.

Our approach is to side-step ongoing improvements to DHTs. Rather than report on the many proposed versions of DHT-based overlays that may quickly become outdated, we study both basic and idealized DHT variants that use global knowledge. This allows us to bound the extent to which performance can be improved as better heuristics are discovered. We use simulation to compare CAN, Chord and Pastry with Narada and NICE when run on the same topologies, at the same scale, and with the same metrics. We also report on power-law random graphs (PLRGs) with flooding based routing, representing Gnutella, to provide another point of comparison. To our knowledge, this is the first "apples to apples" comparison of these approaches.

We find that when configured with the same average out-degree, basic versions of CAN, Chord and Pastry have a latency stretch longer than NICE and Narada by a factor of two or more, depending on the scale. Somewhat surprisingly, all these algorithms showed similar performance in terms of bandwidth hotspots. PLRGs performed better than the basic versions of DHTs in terms of latency stretch due to the use of flooding as the routing mechanism, but poorly in terms of bandwidth hotspots due to highly variable node out-degrees and the same use of flooding. We also find that considerable latency performance gain can be achieved in DHTs with better routing heuristics and topology awareness, though it is more difficult to simultaneously achieve both good latency and good bandwidth performance. Together, these techniques have the potential to bring DHT performance at par with NICE, and thus are a promising direction for future research. As others [4, 19, 16] we find that the hierarchy used to help NICE scale does not significantly degrade its performance as compared to Narada.

The paper proceeds as follows. We describe the relevant overlay algorithms in Section 2. Section 3 discusses the metrics of interest when evaluating overlays, and Section 4 describes our experimental methodology. We present our results in Section 5, discuss related work in Section 6, and conclude in Section 7.

2 Background

This section provides an overview of the overlay construction algorithms that we study. An overlay is built by forming virtual links or tunnels between the participating nodes; a tunnel between a pair of nodes usually traverses multiple links in the underlying network. Given a set of nodes, the goal of an overlay construction algorithm is to select the virtual links and to determine how to route over that topology.

2.1 Measurement-based Overlays

Measurement-based overlays are constructed primarily using active measurements of network properties such as latency between overlay nodes. Several algorithms for building these kind of overlays exist, most of which target multicast services [15, 4, 34, 16, 23, 20, 17, 9]. We study Narada [15] and NICE [4], both of which are optimized for latency.

Narada creates a flat i.e no hierarchal overlay topology that minimizes the latency between nodes while keeping a small number of tunnels per node. This is accomplished by choosing an initial set of tunnels, and periodically adding new tunnels and dropping existing ones. The tunnel addition and deletion process is governed by the utility of the tunnel. The utility metric is computed using the reduction in distance to other nodes the tunnels brings about. This computation requires periodic exchange of routing tables between neighbors and every node probing every other node. This poses a barrier to scalability, but provides all nodes with near global knowledge and leads to an optimized overlay.

To address the scalability problem with a flat measurement based overlay, NICE creates a hierarchy of node clusters. At the bottom of hierarchy, nodes are partitioned into clusters of fixed size. Each cluster has a representative node that lies roughly at the topological center of the cluster. It is determined by having nodes in the cluster probe each other. The representative node of a lower-level cluster is a member in the next level of the hierarchy. This process is recursively repeated, yielding a tree topology. The per node network bandwidth required for overlay maintenance is $O(log(n))$, compared to $O(n)$ in Narada, where n is the number of nodes.

For the purpose of our study, Narada provides a baseline for an overlay with high quality paths at small scale. NICE provides an indication of the performance that can be maintained when additional structure is imposed to scale to larger sizes.

2.2 DHT-based Overlays

DHT-based overlays are constructed using algorithms for distributed hash tables (DHTs) [27, 30, 28, 37, 32]. These algorithms were originally developed to provide highly scalable and fully distributed indexing services for peer-to-peer file sharing. They have since been applied to other services such as multicast [25, 31, 7, 39]. We study CAN [27], Chord [30] and Pastry [28].

In CAN, nodes are mapped pseudo-randomly to a virtual d-dimensional Cartesian space, which wraps around at the edges and has no resemblance to the underlying physical topology. Every node has $2 \times d$ neighbors, corresponding to the adjacent nodes in each dimension. Routing is achieved by following a path through the Cartesian space that increasingly progresses from source to destination. The average path length is $O(dn^{1/d})$, where n is the number of nodes in the overlay and the per node network bandwidth required for protocol maintenance is $O(d)$.

In Chord, every node is pseudo-randomly assigned an m bit identifier. For simplicity of explanation, assume $n = 2^m$, where n is the number of nodes in the overlay. Every node is connected to m neighbors (i.e. $log(n)$ neighbors) with identifiers that are spaced at distances of $2^0, 2^1, 2^2 \ldots 2^{m-1}$ from its identifier in identifier space using modulo arithmetic. A packet from node n_s to node n_d is forwarded to the neighbor of n_s that is arithmetically closest to n_d but less than or equal to n_d. The above process ensures that route between any two nodes has at most $log(n)$ hops. The per node network bandwidth required for protocol maintenance for Chord is $O(log(n))$.

In Pastry, nodes are pseudo-randomly mapped to an m-bit identifier space in base 2^b. The routing table of a Pastry node is a matrix with m/b rows, and 2^b columns. The node in cell (r, c) shares the first r digits with the local node and has the last digit equal to c. Routing is accomplished by each node forwarding a message for key k to the node in its routing table with the longest matching prefix; ties are broken using arithmetic closeness to k. Both the average number of hops required to route a message and per node bandwidth required for overlay maintenance in Pastry is $O(log(n))$.

What we have summarized above are the basic versions of CAN, Chord and Pastry. Definitive descriptions are provided in the papers that we reference. Several modifications have also been proposed [27, 30, 24, 11, 5]. Some of these target performance, while others target application-specific aspects such as data availability, resiliency and hotspot management. Since the primary focus of our work is performance, we study only the performance enhancing heuristics, in Section 4.1, and ignore the rest.

For the purpose of our study, CAN, Chord and Pastry represent the different, major, families of algorithms, all of which are being actively developed [25, 24, 31, 11]. Tapestry [37] is similar to Pastry, and we believe our results are relevant for Tapestry as well.

2.3 Random Power-Law Overlays

We study one more class of overlays to provide another point of comparison: power-law random graphs (PLRGs). It has been observed that the topology of "naturally emerging overlays" such as Gnutella, which are formed when users create links to other nodes, are characterized by a node outdegree distribution that obeys a power law. Routing in such overlays is accomplished through flooding over all links. These overlays are attractive for their simplicity and high tolerance to random failures [29].

3 Overlay Performance Metrics

In this section, we describe the metrics used in our study. Choosing appropriate metrics for comparing overlays is not straightforward because performance depends on the applications. However, the latency overhead of an overlay and its use of network bandwidth is always a concern. For this reason two criteria have been widely used in overlay evaluation: relative delay penalty (RDP) and link stress [15, 4, 25, 31, 17, 5, 7]. We use both of these, along with a third, *Load Balance Ratio*, that measures the distribution of the routing responsibility in the overlay.

Ideally, an overlay would deliver latencies and bandwidths between nodes that match those of the underlying network. However, there is a penalty for routing between overlay nodes. If there are multiple virtual links between two overlay nodes, the path between them through the overlay will be longer than the path through the underlying network. If several virtual links pass over an underlying physical link, the link will experience higher load and in case of multicast the same message may travel over it multiple times. This latency overhead is measured using RDP, and the bandwidth penalty using link stress.

3.1 Relative Delay Penalty

Relative delay penalty (RDP) is a measure of the additional packet delay introduced by the overlay on the delivery of a message between two nodes. It is defined as the ratio of the latency experienced when sending data using the overlay to the latency experienced when sending data directly using the underlying network.

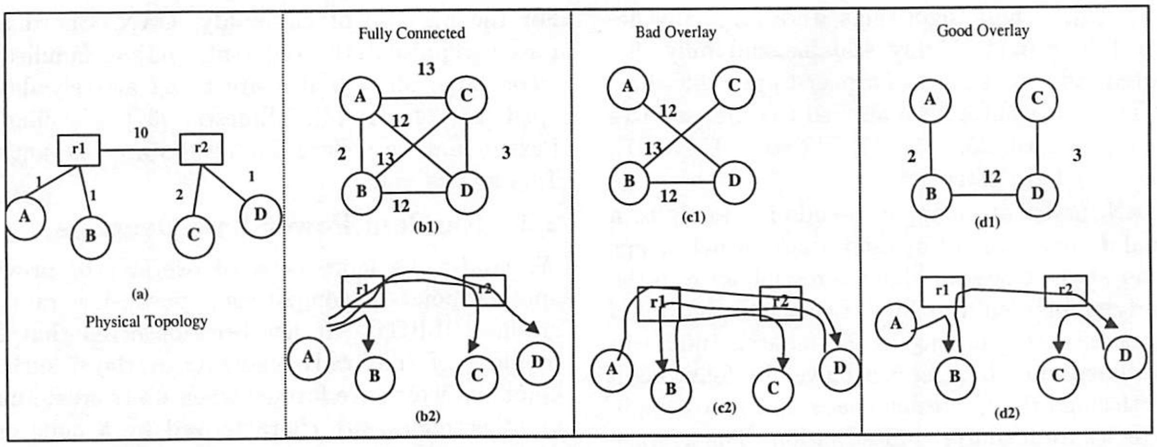


Figure 1: Example overlay topologies. The physical network is shown in the first pane followed by three possible overlays. The logical topology in shown on the top, and the paths taken when A sends a message to B, C and D is shown on the bottom.

As defined, RDP is measured between a pair of nodes and thus provides a set of values for the overlay. To gauge the level of performance of the entire overlay, we use the 90th percentile from the RDP distribution. Various authors have used the same criteria for several reasons [15, 31, 25]. First, the 90th percentile serves to bound the delay multiplier that will be seen by most nodes in practice, whereas the average by itself does not convey any sense of the variation. Second, choosing the 90th percentile rather than the worst case RDP hides sensitivity to simulation parameters. Finally, RDP can easily be very high when the physical latency is small, and using the 90th percentile instead of the worst case RDP filters out these outliers [15].

3.2 Link Stress

Link stress is defined as the number of tunnels that send traffic over a physical link. Links with high stress are potential bandwidth hotspots in the system. In case of unicast, it is a measure of load experienced by the link; more tunnels means higher load. In case of multicast, it becomes a measure of excess bandwidth consumption induced by the overlay; more tunnels means more duplicate messages. For multicast, stress is a function of both the topology and the multicast tree. Two approaches to implement multicast exist – flooding [25] and tree-based [31, 7, 39]. For DHTs and measurement-based overlays, we consider only the latter as it is believed to be more efficient [8]. Here the multicast tree is the union of the unicast routes from the source to all destinations. For PLRGs we use flooding, the default mechanism.

Stress as defined above is a distribution over all physical links for each multicast tree, i.e, per source. To gauge the level of performance of the entire overlay, we use the 90th percentile of the distribution given by the worst stress for each multicast tree. Using worst case distribution conveys a bound on the stress seen by any link, and using the 90th percentile reduces the sensitivity to simulation parameters as before.

An ideal overlay should have both low RDP and low stress. Unfortunately, it may be difficult to simultaneously achieve both objectives. For instance, consider the physical network and three possible overlays topologies shown in Figure 1. Figure 1(b1) shows a fully connected overlay topology, in which the RDP between all pairs is 1, but the stress on links close to the end hosts is high. To communicate with B, C, and D simultaneously, A must send three packets over its physical access link, leading to a stress of 3. Next consider the overlay topology in Figure 1(c1), which has low stress on links close to the end hosts. In this case, all overlay tunnels go over the high latency physical link joining **r1** and **r2**. This leads to high RDP between most pairs. Figure 1(d) shows a good overlay with both acceptably low RDPs and low stresses.

3.3 Load Balance

In an overlay, nodes also act as routers and forward packets between other nodes. Ideally an overlay would not place a much higher forwarding load on one node compared to other nodes[1]. Otherwise, with increasing workload the overloaded node would soon become a hotspot for the system, leading to

[1]Here we are assuming homogeneous nodes. Although this may not be a realistic assumption in practice, existing protocols are not designed to take into account heterogeneity of members [26]. Any variation in forwarding load is thus unintended.

degraded performance. For example, logical topologies such as a star or a tree are not well balanced compared to a ring topology because they have key nodes that are used for most point-to-point communications.

To measure the distribution of the forwarding load, we define a metric called the *Load Balance Ratio*, which is computed as follows. For each node in the overlay, the routing load is the number of source-destination pairs between which it forwards messages. Load balance ratio is the ratio of the maximum routing load to the median routing load. This measures how much worse the maximally loaded node is compared to the halfway loaded node. This metric is less relevant for multicast where all nodes forward one message for a given source. It is important when overlays are used more generally for multiple point-to-point communications.

For the purpose of our study, load balance ratio exposes how performance-based routing concentrates traffic in non-uniform ways. One of the favorable arguments behind DHT-based approaches is that they are better load balanced because of their regular geometric structure and use of randomization. But heuristics that improve performance can interfere with this. We also note that load balance distinguishes topologies and routing protocols that are only suited for multicast from those that are more widely applicable. For example, protocols such as NICE that use hierarchy are good for multicast but would place extremely high load on nodes high in the tree if used for general unicast communications.

3.4 Other Considerations

There are several other performance measures that we do not explore in this paper. We do not measure the overhead of the overlay protocols, either in terms of the amount of state or traffic that is needed to maintain the overlay. It is well known that pure DHT algorithms make only local measurements and scale extremely well, while schemes such as Narada perform global measurements and scale relatively poorly. Our emphasis instead is on the performance levels that can be achieved at a given scale for which the overhead of the algorithms under study has been deemed acceptable. For example, NICE is able to scale to 1000s of nodes (because its overhead is logarithmic with overlay size), while Narada is not. Thus, for overlays this large, NICE is our only option among measurement-based overlays for comparison with DHTs.

We also do not measure protocol dynamics, such as maintaining connectivity in face of failures. Overlays are expected to operate with members leaving dynamically, and different overlay construction algorithms may be disrupted in different ways; in the extreme, partitions are possible. While these dynamic properties are important, our focus is to first understand the static performance potentials of the different algorithms.

4 Methodology

In this section, we describe our experimental methodology. We first describe the variations of the DHT-based algorithms that we compare. We then describe our simulation set-up.

Recall that our goal is to understand whether DHT-based overlays will be able to match the level of performance of measurement-based overlays, which are constructed specifically to provide good quality paths. One difficulty is that different heuristics are continually being proposed to enhance the performance of DHT-based overlays [24, 27, 38, 31, 32, 5], and we do not wish our results to quickly become irrelevant by being tied too closely to specific heuristics. Instead, we side-step this race by taking advantage of simulation as a tool to report on the performance of both the current versions and idealized versions that enhance performance by using global knowledge.

4.1 DHT Heuristics

We study performance-enhancing variations to DHTs along two dimensions – those that attempt to find better paths over a given overlay, and those that construct the overlay itself in a topology-aware manner. We describe each in turn.

4.1.1 Routing Heuristics

In the simplest version of DHTs, routing proceeds using only the geometric properties of the overlay algorithm. We refer to this as the *Base* variant. It is possible to improve performance by routing across the overlay in a manner that takes latency into account. The structure of the overlay itself remains unaffected. We study the heuristic specified in [27] for CAN and [11] for Chord. Here, the chosen next hop is the one out of all possible neighbors that results in the maximum progress towards the destination, where progress is defined as the ratio of the distance in identifier space to the physical latency. We refer to these versions of CAN and Chord as the *Proximity* variants. In Pastry, the next hop is unique and therefore it does not have a corresponding proximity routing variant. Comparison between *Base* and *Proximity* shows the value of the current routing heuristics.

Other heuristics have been proposed, e.g., smallest physical latency for CAN as described in [8]. However, rather than simulate all the different heuristics we can find, we stick with the above as a yardstick (since it applies to both CAN and Chord and is the subject of most published results) and define a new variant intended to provide an upper bound on how well any future heuristic can perform. This variant, *Shortest-Path,* comes from the observation that the maximum gain achievable by any routing heuristic is that of shortest path routing (implemented using a distance vector or link state algorithm). This algorithm requires global knowledge, unlike proposed heuristics, and so may not be a good choice at large scales. However, it can readily be computed in our simulation setting to provide a bound on the performance of better heuristics that will inevitably be proposed. That is, comparison between *Proximity* and *Shortest-Path* shows how much room routing heuristics have for improvement. Another advantage is that the *Shortest-Path* variant is independent of the actual overlay algorithm and therefore it applies to all three DHTs.

4.1.2 Topology Awareness

In the simplest version of DHTs, both the mapping of nodes to identifiers and the choice of tunnels (when multiple choices are possible as in Pastry) is pseudo-random to provide a well-balanced structure. This overlay construction process, which we refer to as the *Random* variant, is unaware of the underlying topology.

In a topology aware overlay, heuristics are used to create an overlay that reflects the underlying network topology. The intuition is that if the overlay and the underlying network topology closely mirror one another, then the overlay paths will closely follow network paths and good performance will result. We now describe how to achieve topology awareness in Pastry, CAN and Chord.

In Pastry topology awareness can be achieved through neighbor selection [5]. A routing table entry can be filled by any node that matches the criteria for that cell. Topology aware construction chooses the closest such node. Achieving perfect awareness using this method requires global knowledge, though reasonable approximations that rely only on limited information exchange are possible [5]. In this paper, we only consider the global knowledge approach.

The above neighbor-selection approach is not applicable to CAN and Chord because in both these algorithms topology is completely defined once the identifier assignment is done. Instead topology awareness is achieved through intelligent identifier assignment. For instance, identifier assignment based on proximity to landmarks has been proposed as a heuristic [24] to achieve a topology aware mapping in CAN.

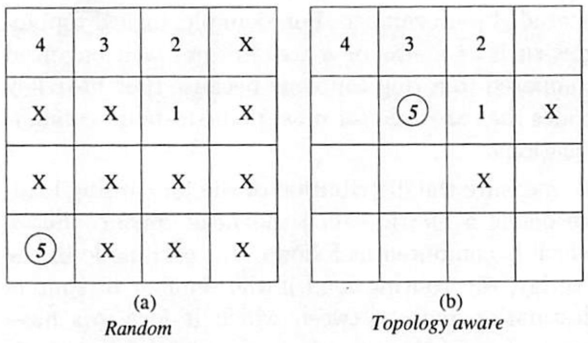


Figure 2: Topology awareness in a 2-d CAN. A new node 5, which is physically closest to 1, joins the overlay. X denotes the possible assignments for 5 in case of *Random* and *Topology-Aware* overlay construction. The *Topology-Aware* case yields an assignment in which 5 is geometrically close to 1.

However, it is generally the case that heuristics for topology-aware identifier assignment are not as well defined nor studied as heuristics for improving routing. To understand how much topology awareness itself can improve performance, assuming that good heuristics will be found, we would like to use an analogue to our *Shortest-Path* variation above, rather than simulate many possible heuristics. However no such globally optimal assignment has been defined for either CAN or Chord. So, we use a greedy assignment based on global knowledge to construct a good assignment. Our greedy assignment[2] rule works as follows: the identifier of a new node joining the overlay is chosen so that it is a neighbor of the node that is closest to it in the underlying network. That is, the rule makes the overlay neighbors of a node similar to the neighbors of the node in the underlying topology. Figure 2 illustrates the concept for a 2-dimensional CAN. This also requires global knowledge (or at least a knowledge of distances to all nodes in the vicinity) and so may not be a good implementation choice at large scales. Interestingly, a similar heuristic has been proposed in parallel with our work [32].

4.2 Simulation Setup

Since our primary interest is in understanding performance under static environments, we simulate

[2]We believe that our approach based on global knowledge would yield a better mapping than using a fixed number of landmarks. The performance of the latter is also sensitive to the choice and number of landmarks.

overlay construction using centralized algorithms. The only exception is Narada, which we simulated using an event-driven simulator because Narada does dynamic evaluations to improve the overlay over time. To generate PLRGs overlays, we first generated power-law topologies using Brite [21], and then randomly mapped the nodes in this graph to the nodes in the overlay. For DHTs, pseudo-random hash functions are used to provide a well-balanced structure. However, since we are using simulation we created well-balanced structures by uniformly partitioning the space directly. This is a conservative simplification that is consistent with our goal of presenting the DHT-based overlays in their best light.

An important parameter that affects the performance is the average out-degree (the number of neighbors of a node). It determines the number of links in the overlay and therefore directly affects performance. For example, an overlay with more links will have lower RDPs because of shorter paths but higher stress because more overlay links traverse a physical link. We study the effect of out-degree in Section 5.4. The average out-degree is a configurable parameter in all overlays except Chord. In Chord the outdegree is $log_2(overlay\_size)$. In Pastry the average degree depends on b and varies as $b log_{2^b}(overlay\_size)$. We fixed b as 1 to make Pastry's average degree comparable to that of Chord. The values of degree we study range from 4 to 12.

We used the transit-stub model of GT-ITM [35] to generate the physical network topology. GT-ITM also assigns latencies to links in the physical topology. Additional nodes were attached to the stub nodes to represent hosts connected to lower level routers. Overlay nodes were picked randomly from these hosts.

The overlay algorithms that we compare are summarized in Table 1. We study one version of each of Narada [15], NICE [4], and PLRG. Narada and NICE build their topologies using latency measurements as described earlier, and they use shortest path routing using latency as the metric. PLRG forms a random topology where the node outdegree distribution follows a power law, and nodes flood over this topology; this represents Gnutella-like overlays.

For each parameter setting we ran 9 simulations, three different physical network topologies, each with three random seeds. Each topology had 4,040 backbone nodes and 20,200 stub nodes. Simulated overlay sizes were varied from 64 to 4096.

	Topology Awareness	Routing	Degree
Narada	-	*Shortest-Path*	$4-12$
NICE	-	*Shortest-Path*	$4-12$
CAN	*Topology-Aware, Random*	*Shortest-Path, Proximity, Base*	$4-12$
Chord	*Topology-Aware, Random*	*Shortest-Path, Proximity, Base*	–
Pastry	*Topology-Aware, Random*	*Shortest-Path, Base*	–
PLRG	-	*Flooding*	$4-12$

Table 1: Summary of simulated algorithms.

5 Results

We now present the results of comparing different protocols and their variants on each of our three performance metrics. We first consider a comparison using an out-degree of 10 for all overlays except Chord and Pastry. This out-degree was found to be a good representative by experimenting with different out-degrees. The effect of out-degree on performance is described in Section 5.4.

5.1 RDP

This section investigates the latency stretch of various overlays. We start by studying the impact of heuristics in DHT-based overlays, and then compare all the overlays.

5.1.1 Effect of Heuristics

Figures 3 and 5 show the impact of routing heuristics in CAN on top of an overlay with *Random* and with *Topology-Aware* overlay construction, respectively. They plot the 90th percentile RDP for the *Base* version of CAN, CAN with *Proximity* routing, and CAN with *Shortest-Path* routing (the best possible performance for a given overlay). Similar results were obtained using 50th percentile and 95th percentile RDP and are not shown. Figures 4 and 6 show the same results for Chord. Results from all 9 simulations are presented to show the variance and the line is drawn through the average. This representation is used in all plots unless otherwise specified.

For both CAN and Chord, improved routing brings about significant improvement in RDP. *Proximity* routing does quite well; it reduces the RDPs to roughly halfway between *Base* and *Shortest-Path*

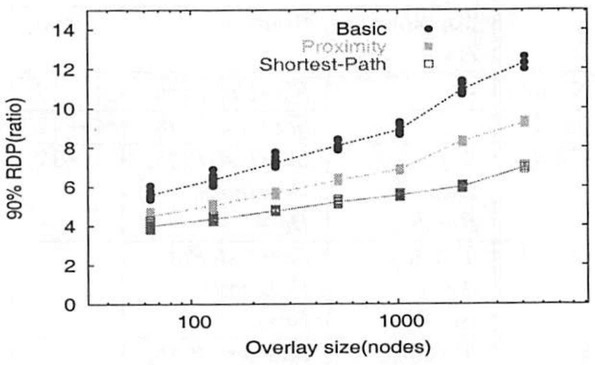


Figure 3: Effect of routing heuristics on RDP in CAN with *Random* overlay construction.

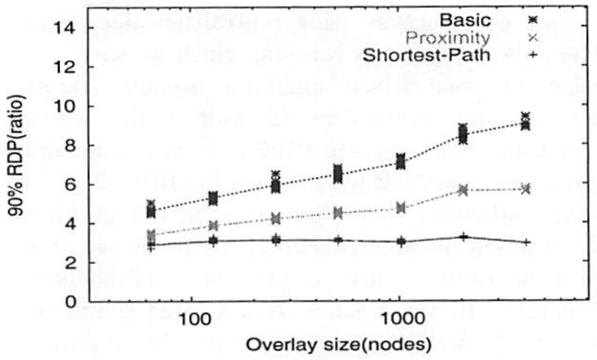


Figure 5: Effect of routing heuristics on RDP in CAN with *Topology-Aware* overlay construction.

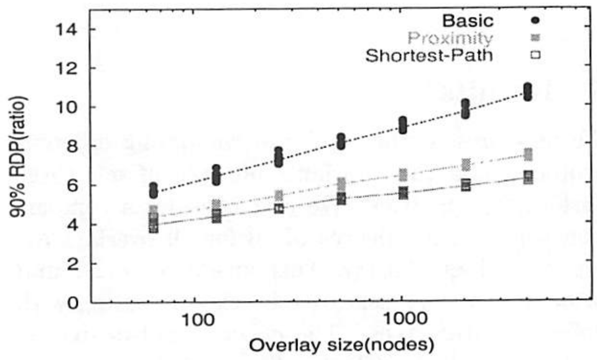


Figure 4: Effect of routing heuristics on RDP in Chord with *Random* overlay construction.

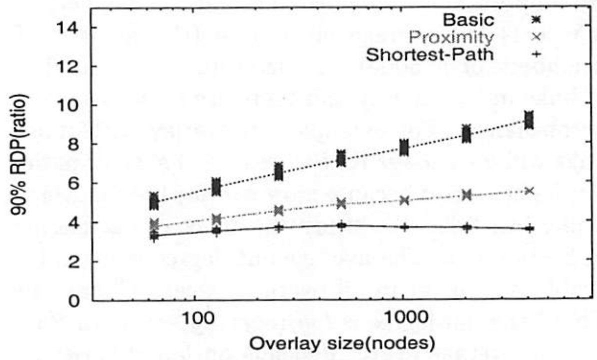


Figure 6: Effect of routing heuristics on RDP in Chord with *Topology-Aware* overlay construction.

	Topology Awareness	
Routing	*Random*	*Topology-Aware*
Base	**8.89**	7.05
Proximity	6.87	4.71
Shortest-Path	5.55	**3.02**

Table 2: Effect of heuristics on 90th percentile RDP for CAN with 1024 nodes

routing. However, it does not match the performance of *Shortest-Path* routing because while the former is a greedy decision at each hop, the latter computes globally optimal paths. The improvement in Chord using *Proximity* routing is slightly greater because as we increase the overlay size, the number of choices for the next hop increases, leading to better paths. As a result, *Proximity* routing comes closer to *Shortest-Path* routing in Chord than in CAN.

To understand the effect of combining the different heuristics, we tabulate the 90th percentile RDP for CAN with 1024 nodes in Table 2. Observe that the effect of heuristics compose and the best performance is achieved by enabling both *Shortest-Path* routing and *Topology-Aware* overlay construction. This improvement is substantial, from 8.89 to 3.02 (almost a 70% reduction), and indicates the potential for improvement through more practical heuristics. By looking across columns in the table, we deduce that being topology aware by itself brings about a significant performance gain.

Figure 7 shows the effect of both topology awareness and routing heuristics on RDP for Pastry. As before the heuristics lead to substantial improvement.

5.1.2 Comparing All Protocols

We try to answer two questions now:

1. How do DHTs with only scalable heuristics (such as *Proximity* routing) compare to measurement-based overlays, especially NICE since it has similar scalability?

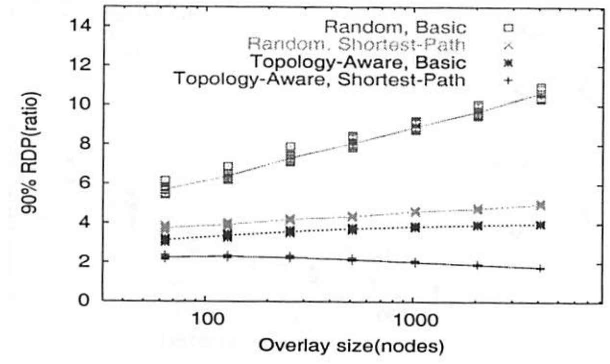


Figure 7: Effect of heuristics on RDP in Pastry.

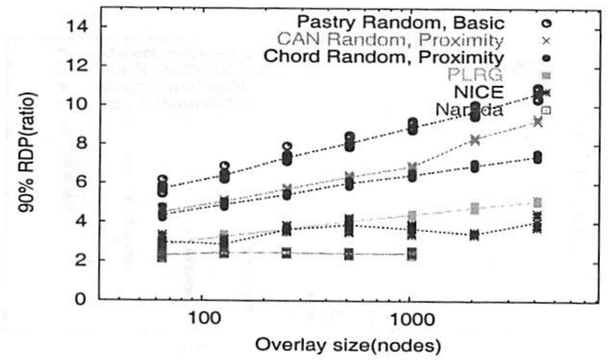


Figure 8: Variation of RDP with size. Simple practical versions of DHTs are shown – CAN and Chord with *Random* overlay construction and *Proximity* routing, and Pastry with *Random* overlay construction and *Base* routing.

2. How do DHTs compare to the other approaches when *Shortest-Path* routing along with *Topology-Aware* overlay construction is implemented?

Figure 8 shows the 90th percentile RDP as a function of overlay size for different protocols[3]. *Proximity* routing is used for CAN and Chord with *Random* overlay construction. For Pastry we show *Random* overlay construction with *Base* routing. These represent simple, efficient, and practical versions of these overlays. A distributed implementation of topology awareness is shown to be a reasonable approximation in [6]. We believe that by the same token, similar implementations are possible for CAN/Chord because a new node being able to find the closest live node is a key assumption in both scenarios. Instead of assuming this to be a fact and for

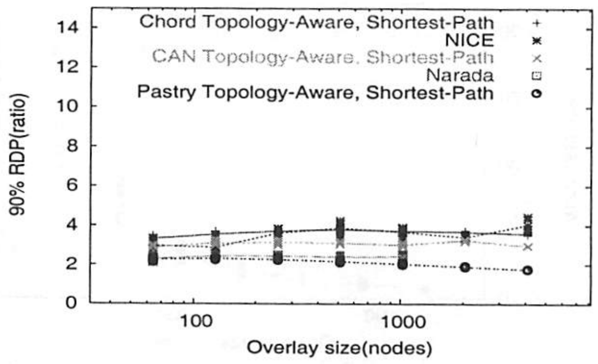


Figure 9: Variation of RDP with size for various overlays. Idealized versions of DHTs – with *Topology-Aware* overlay construction and *Shortest-Path* routing – are shown.

fairness of comparison, we chose to use the *Random* overlay construction for all DHTs.

We make the following observations:

• The difference in RDP between NICE and the DHTs is large and grows with the overlay size. It is a factor of two for 1024 nodes.

• Flooding over PLRGs performs well, because with flooding the shortest path between nodes is taken. This also indicates that without heuristics DHT overlays are not better than random overlays from a latency perspective.

• All DHTs have similar RDPs. The slightly worse performance for Pastry stems from the absence of a *Proximity* routing equivalent, and the slightly better performance of Chord for overlay sizes greater than 1024 stems from a higher average out-degree in Chord compared to CAN beyond this size,[4] which leads to shorter paths.

• The performance of NICE does not deteriorate with increasing overlay size even though the levels of hierarchy increase. Further, it is not much worse than that of Narada. Our findings agree with those of the authors of NICE [4].

Figure 9 compares DHTs with *Topology-Aware* overlay construction and *Shortest-Path* routing with all other protocols. This represents the maximum performance one can achieve from a latency perspective for these overlays. There is no real qualitative difference between performance in this case. This is encouraging because it shows that RDPs comparable to measurement-based overlays can be achieved using improved versions of DHT-based overlays.

[3]For Narada, results are not shown for overlay sizes greater than 1024 nodes because simulation times were on the order of days.

[4]Average degree for Chord is $log(n)$ – for 1024 nodes average degree is 10 which is the same as the average degree configured for CAN.

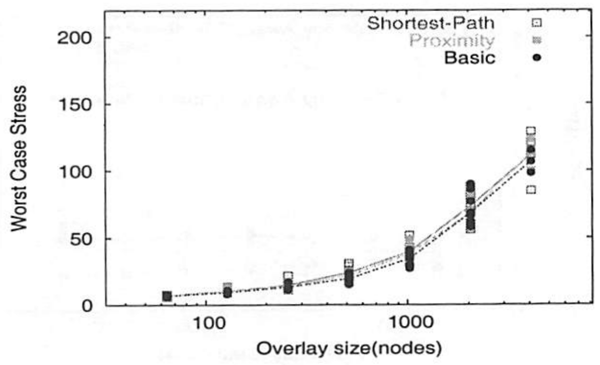


Figure 10: Effect of routing heuristics on link stress in CAN with *Random* overlay construction.

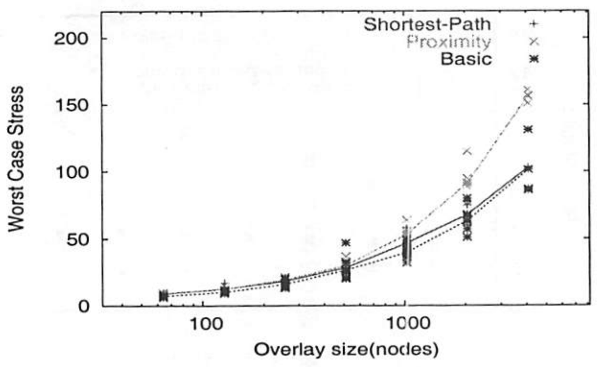


Figure 11: Effect of routing heuristics on link stress in CAN with *Topology-Aware* overlay construction.

5.2 Stress

We now investigate how the various protocols compare on the metric of the worst case stress, and the impact on stress of RDP-enhancing heuristics in DHT-based overlays.

5.2.1 Effect of Heuristics

Figures 10 and 11 show stress as a function of overlay size for CAN with *Random* and *Topology-Aware* overlay construction, respectively. The three lines correspond to the three routing variants - *Base*, *Proximity* and *Shortest-Path*. The results for Chord were similar, and have been omitted due to space constraints.

Somewhat surprisingly, improved routing does not have much negative impact on stress. With improved routing, we expected stress near a few well-placed nodes to go up. This is evident to some degree in the fact that the worst case stress does go up slightly. But at the same time, this increase is reasonably countered because improved routing leads to shorter paths, which means that fewer links are traversed. By comparing the results across *Random* (Figure 10) and *Topology-Aware* (Figure 11), we can also see that topology awareness has little impact on stress.

Figure 12 shows the effect of heuristics on link stress for Pastry. As for CAN, both routing heuristics had no significant impact on stress. However, topology awareness with *Base* routing had much worse (almost factor of 2) stress values. A key difference between the topology aware overlay construction mechanisms of Pastry and CAN/Chord is that in Pastry a few well placed nodes can be the neighbors of many nodes, whereas in CAN/Chord a node can be a neighbor of only a small number of other nodes. This has a direct consequence on link stress and load balancing. Hence, our results should be interpreted as highlighting the difference between the two methods to achieve topology-awareness – choosing node identifier assignment and choosing neighbors.

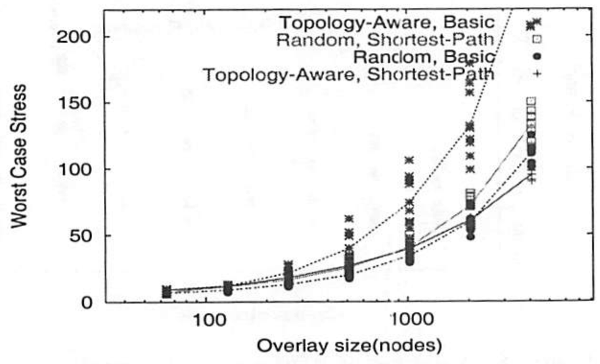


Figure 12: Effect of heuristics on link stress in Pastry.

5.2.2 Comparing All Protocols

Figure 13 shows how stress varies with size for different algorithms. Since the performance of various variants of DHTs was largely similar, we show the stress only for the best version, *Topology-Aware* overlay construction with *Shortest-Path* routing. The striking artifact in the graph is that PLRG has very high stress, more than five times worse than the other protocols for 1024 nodes. This is due to its use of flooding as the routing mechanism, and an uneven out-degree distribution among nodes. All other protocols exhibit similar stress values, although for large overlay sizes (over 2048 nodes) NICE has slightly smaller values. Large group sizes have higher density for a fixed topology size, which has the tendency to increase the stress on the backbone links in DHTs. In this situation, the clustering

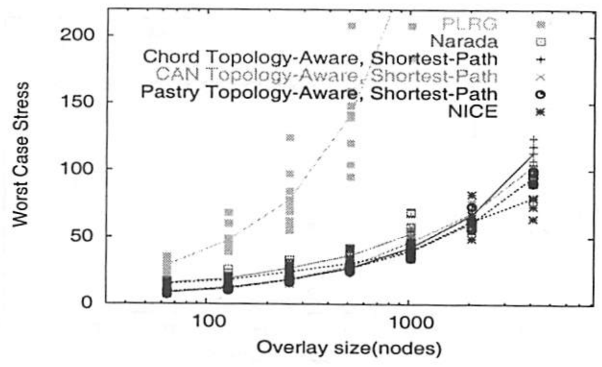


Figure 13: Variation of stress with size for various overlays. Variants of DHT with *Topology-Aware* overlay construction and *Shortest-Path* routing are shown.

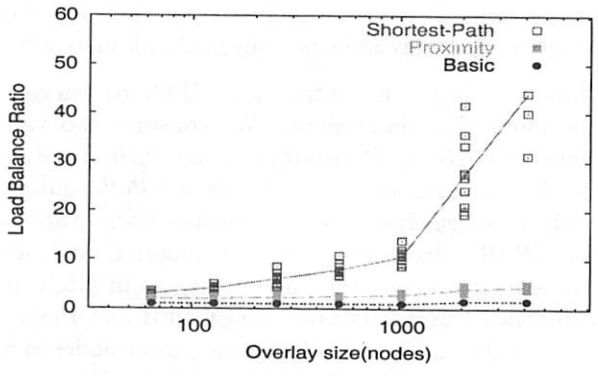


Figure 14: Effect of routing heuristics on load balancing in CAN with *Random* overlay construction.

of close nodes in NICE helps to reduce the stress on the backbone links.

In summary, for similar average out-degrees all protocols exhibited similar worst case stress properties. In case of CAN and Chord heuristics have no significant impact on stress properties. For Pastry, however, *Topology-Awareness* with *Base* routing had significantly higher stress than other variants.

5.3 Load Balancing

We now study the load balancing properties.

5.3.1 Effect of Heuristics

Figure 14 shows the load balancing ratio for different overlay sizes for CAN with *random* overlay construction. As before, the three lines correspond to the three routing variants. With *Base* routing, the load is balanced very evenly, with the ratio between one and two. This is a direct consequence of the regular structure and the random overlay construction in the DHT algorithms. With the *Proximity* routing, load balancing deteriorates slightly but grows very slowly with the overlay size. However, there is a significant degradation with *Shortest-Path* routing because there exist well-placed nodes in the topology that have a low latency to many nodes, and therefore are used very often independent of their node-ID. This raises the issue of the value of a routing heuristic that mimics *Shortest-Path* routing if balanced load is desired.

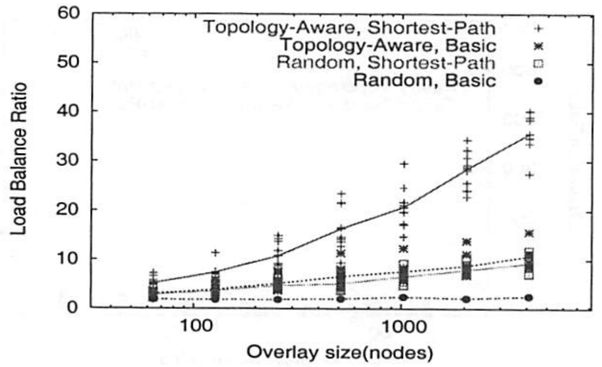


Figure 15: Effect of heuristics on load balancing in Pastry.

We observed similar behavior for CAN with *Topology-Aware* overlay construction because being topology aware only changes the relative assignments and does not modify the DHT structure itself. The results for Chord were also similar; we omit them due to space constraints.

Figure 15 shows the load balancing ratio for different heuristics. As in CAN we find significant degradation with *Shortest-Path* routing. We can also see that in the case of Pastry *Topology-Awareness* caused the load balance to deteriorate. This is for same reasons, as for higher link stress, mentioned in Section 5.2.1.

5.3.2 Comparing All Protocols

Figure 16 shows the load balancing ratio for all the protocols. The *Shortest-Path* routing variant for DHTs is shown, as it had the highest load. Note that the scale of y-axis differs from that in Figure 14. We can see that NICE has an extremely high load balancing ratio (two orders of magnitude for overlays bigger than 1024). This is because in the NICE hierarchy, the root of hierarchy is responsible for forwarding all packets whose source and destination lie in different sub-trees. On the other hand, Narada, a measurement-based overlay with no hierarchy, performs similarly to CAN and Chord with *Shortest-Path* routing. PLRG too has a high load balancing

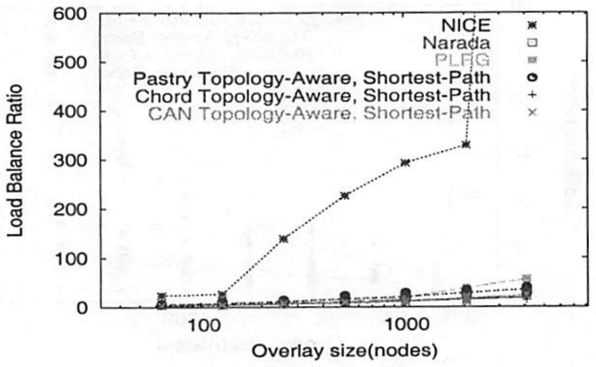


Figure 16: Variation of load balancing ratio with size for various overlays. Variants of DHTs with *Topology-Aware* overlay construction and *Shortest-Path* routing are shown.

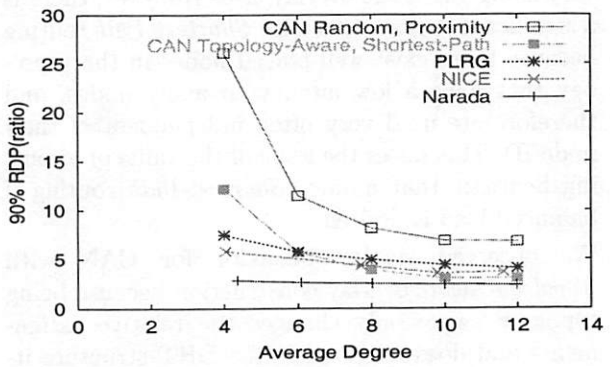


Figure 17: Effect of out-degree in various overlays. Two variants of CAN are shown – *Random* overlay construction with *Proximity* routing and *Topology-Aware* overlay construction with *Shortest-Path* routing.

ratio because of a highly uneven degree distribution. Note that the relatively sharp increase in NICE at some points (256 and 4096) is a quantization effect due to formation of unbalanced trees.

In summary, our results point at the limitations of hierarchy for applications involving general unicast communications. DHT routing heuristics that mimic *Shortest-Path* routing can have significant negative impact on load balancing and thus may not be a suitable choice for some applications. Although *Topology-Aware* overlay construction by itself does not degrade load balancing in CAN/Chord, it degrades it in the case of Pastry.

5.4 Effect of Out-Degree

In this section we study the effect of increasing the average out-degree of nodes in various overlays. Intuitively, increasing out-degree will decrease RDP because it reduces the number of overlay hops. But at the same time, stress could increase because the number of overlay links per physical link increases.

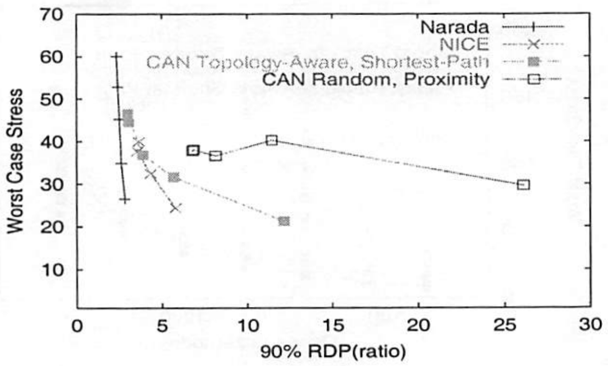


Figure 18: Variation of link stress with RDP. Two variants of CAN, same as those in Figure 17, are shown.

Node out-degree is controlled in CAN by varying the number of dimensions. We consider two versions of CAN: *i)* *Proximity* routing with *Random* overlay construction; and *ii)* *Shortest-Path* routing with *Topology-Aware* overlay construction. Narada and PLRG algorithms can be configured with an average out-degree, and the out-degree in NICE is controlled through cluster size. Chord and Pastry are not shown because the out-degree of nodes can not be varied independent of the overlay size.

The results in this section are presented for a fixed overlay size of 1024 nodes and only averages over simulations are plotted. Figure 17 shows the effect of increasing average out-degree on RDP for different protocols. Two observations can be made. First, there is a sharp reduction in RDP for CAN variants as the average degree is increased. For other protocols also, the RDP decreases with increasing degree but the gain is much less. Second, NICE was able to achieve good RDPs with a much lower average out-degree compared to CAN with *Proximity* routing over *Random* overlay construction.

We now explore the trade-off between RDP and link stress. Figure 18 shows the relationship between stress and RDP. It plots the measured stress for a given RDP obtained by varying the node out-degree (Figure 17). The worst case stress for PLRGs was extremely high, and hence has not been shown in this graph. All overlays exhibit the basic tradeoff between RDP and link stress, though to varying degrees. By increasing average out-degree, RDP can be reduced, but that reduction comes at the cost of higher stress. However, note that the CAN variant with *Proximity* routing over *random* overlay construction lies further away from the origin than NICE. This means that it is harder to simultaneously achieve both low stress and low RDP in this variant of CAN than in NICE. At the same time CAN with *Shortest-Path* routing and *Topology-Aware* overlay construction performs comparably to NICE, which again points favorably towards the potential of optimization in DHT-based approaches.

6 Related Work

Many overlay construction schemes have been proposed, both measurement-based [15, 4, 20, 2, 23, 9, 34, 17] and DHT-based [27, 30, 28, 37]. Much ongoing work aims to improve the performance of DHT-based approaches [5, 36, 38, 24, 33, 31, 11] and the scalability of measurement-based approaches [4, 20], as well as look at how DHT-based approaches can provide multicast and other services [25, 7, 39, 31]. However, there has been very little work on studying how these different overlay algorithms compare to each other. Our work and that of a few other researchers are first steps in this direction.

Castro et.al compare the performance of tree building and flooding on top of CAN and Pastry [8]. They find that flooding has high overhead compared to tree-based approaches. This is consistent with our results, where we found that flooding on top of PLRGs has much higher overhead compared to tree-based protocols. A qualitative comparison of various overlay protocols is provided in [3]. Our focus, however, is on empirical evaluation under identical environments.

Finally, Ratnasamy et. al outline the challenges facing DHT-based overlays [26]. They identify performance with respect to latency as a key open issue. Our work indicates that with good routing heuristics and topology-awareness, DHT-based overlays can match the performance of measurement-based overlays.

7 Conclusions and Future Work

In this paper we studied the performance potential of DHT-based overlays at moderate scale (1000s of nodes) where they represent an alternative to measurement-based overlays for multicast and other services. We used simulation to compare basic and ideal DHT-based overlay protocols with measurement-based protocols, at the same scale, using the same topology, and with the same performance metrics. Specifically, we compared CAN, Chord and Pastry with Narada and NICE, as well as power-law random graphs.

Our key findings are:

• For the same average out-degree, basic versions of DHTs have a latency stretch that is longer than NICE and Narada by a factor of two or more, depending on the size of the overlay. The performance in terms of bandwidth hotspots is similar however.

• Considerable performance gains in latency can be achieved in DHTs with better routing heuristics and with topology-aware overlay construction. These heuristics had no substantial adverse effect on link stress, except when the choice of tunnels was directly sensitive to latency, as in Pastry. However, the heuristics (and especially the routing heuristics) do lead to relatively higher load on some nodes.

• As others [4, 16, 19], we found that the hierarchy in NICE does not significantly degrade performance compared to Narada from a latency perspective. The effect of hierarchy on bandwidth hotspots was minimal too. We also note that power-law random graphs had a latency stretch that is competitive with that of NICE, but performed poorly in terms of bandwidth hotspots.

In summary our findings indicate that DHT-based overlays are a promising direction, with the potential to achieve not only scale but good levels of performance. Better heuristics that come closer to achieving this potential without sacrificing scalability are the subject of other, ongoing research.

There are also directions in which we hope to extend our study. The optimizations we considered in this paper are biased towards latency. We do not yet know if bandwidth heuristics (we are not aware of any at present) can deliver levels of bandwidth that are comparable to that of bandwidth optimizing protocols such as Overcast [17] and the enhanced version of Narada [14]. Performance in dynamic environments is another area worthy of exploration.

Acknowledgements

We would like to thank Gaetano Borriello and Steve Gribble for discussions during early stages of this work.

This work was supported by DARPA grant F30602-98-1-0205.

References

[1] Akamai. http://www.akamai.com.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *ACM SOSP*, 2001.

[3] S. Banerjee and B. Bhattacharjee. A comparative study of application layer multicast protocols. In *Submitted for review*, 2002.

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *ACM SIGCOMM*, 2002.

[5] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *FuDiCo*, 2002.

[6] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *Technical Report MSR-TR-2002-82*, 2002.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. In *IEEE JSAC*, 2002.

[8] M. Castro, *et al.* An evaluation of scalable application-level multicast built using peer-to-peer overlay networks. In *IEEE INFOCOM*, 2003.

[9] Y. Chawathe, S. McCanne, and E. Brewer. An architecture for internet content distribution as an infrastructure service, Unpublished work, 2000.

[10] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Design Issues in Anonymity and Unobservability*, 2000.

[11] F. Dabek, *et al.* Wide-area cooperative storage with cfs. In *ACM SOSP*, 2001.

[12] Digital Island. http://www.digitalisland.com.

[13] Gnutella. http://www.gnutella.com.

[14] Y. hua Chu, S. G.Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the Internet using an overlay multicast architecture. In *ACM SIGCOMM*, 2001.

[15] Y. hua Chu, S. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS*, 2000.

[16] S. Jain, R. Mahajan, D. Wetherall, and G. Borriello. Scalable self organizing overlays. Tech. Rep. UW-CSE 02-06-04, University of Washington, 2002.

[17] J. Jannoti, *et al.* Overcast: Reliable multicasting with an overlay network. In *OSDI*, 2000.

[18] Kazaa. http://www.kazaa.com.

[19] D. Kostic and A. Vahdat. Latency versus cost optimizations in hierarchical overlay networks. Tech. Rep. CS-2001-04, Duke University, 2001.

[20] M. Kwon and S. Fahmy. Topology-aware overlay networks for group communication. In *ACM NOSSDAV*, 2002.

[21] A. Medina, I. Matta, and J. Byers. On the origin of power laws in Internet topologies. Tech. Rep. 2000-004, Boston University, 2000.

[22] Napster. http://www.napster.com.

[23] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, pp. 49–60. San Francisco, CA, USA, 2001.

[24] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *INFOCOM*, 2002.

[25] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content-addressable network. In *NGC*, 2001.

[26] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. In *IPTPS*, 2002.

[27] S. Ratnasamy, *et al.* A scalable content addressable network. In *ACM SIGCOMM*, 2001.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[29] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, 2002.

[30] I. Stoica, *et al.* Chord: A peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM*, 2001.

[31] I. Stoica, *et al.* Internet indirection infrastructure. In *ACM SIGCOMM*, 2002.

[32] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. In *Proceedings of HotNets-I*, 2002.

[33] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. Research Report RZ-3436, IBM, 2002.

[34] Yoid: Your own Internet distribution. http://www.isi.edu/yoid/.

[35] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, 1996.

[36] B. Y. Zhao, A. Joseph, and J. Kubiatowicz. Locality-aware mechanisms for large-scale networks. In *FuDiCo*, 2002.

[37] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, UCB, 2001.

[38] B. Y. Zhao, *et al.* Brocade: Landmark routing on overlay networks. In *IPTPS*, 2002.

[39] S. Q. Zhuang, *et al.* Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV*, 2001.

Moving Edge-Side Includes to the Real Edge—the Clients

Michael Rabinovich† Zhen Xiao† Fred Douglis‡
Chuck Kalmanek†

†*AT&T Labs – Research*
‡*IBM T.J. Watson Research Center*

Abstract

Edge-Side Includes (ESI) is an open mark-up language that allows content providers to break their pages into fragments with individual caching characteristics. A page is reassembled from ESI fragments by a content delivery network (CDN) at an edge server, which selectively downloads from the origin content server only those fragments that are necessary (as opposed to the entire page). This is expected to reduce the load and bandwidth requirements of the content server.

This paper proposes an ESI-compliant approach in which page reconstruction occurs at the browser rather than the CDN. Unlike page assembly at the network edge, CSI optimizes content delivery over the last mile, which is where the true bottleneck often is. We call the client-based approach *Client-Side Includes*, or CSI.

1 Introduction

As the use of the Internet increases, caching is an important tool in coping with the rate of requests to Internet servers. Caching can be client-centric (proxy caching) or server-centric (reverse proxy caching or CDNs). Regardless, a significant limitation of caching is that it is mostly oriented toward *static* content. This limitation has been recognized and there have been a number of approaches to extend caching to handle other types of content.

These approaches include specialized tools to generate pages with dynamic content on a client (e.g., HPP [6] and <bigwig> [3]) or a proxy cache (e.g., Active Cache [4] and CONCA [19]); application distribution networks, which run complete applications at the edge of the network (e.g., Ejasent [9] and ACDN [13]); and Edge-Side Includes (ESI) [10], which builds pages, from component pieces (known as *fragments*) specified within an XML template, in servers at the network edge.

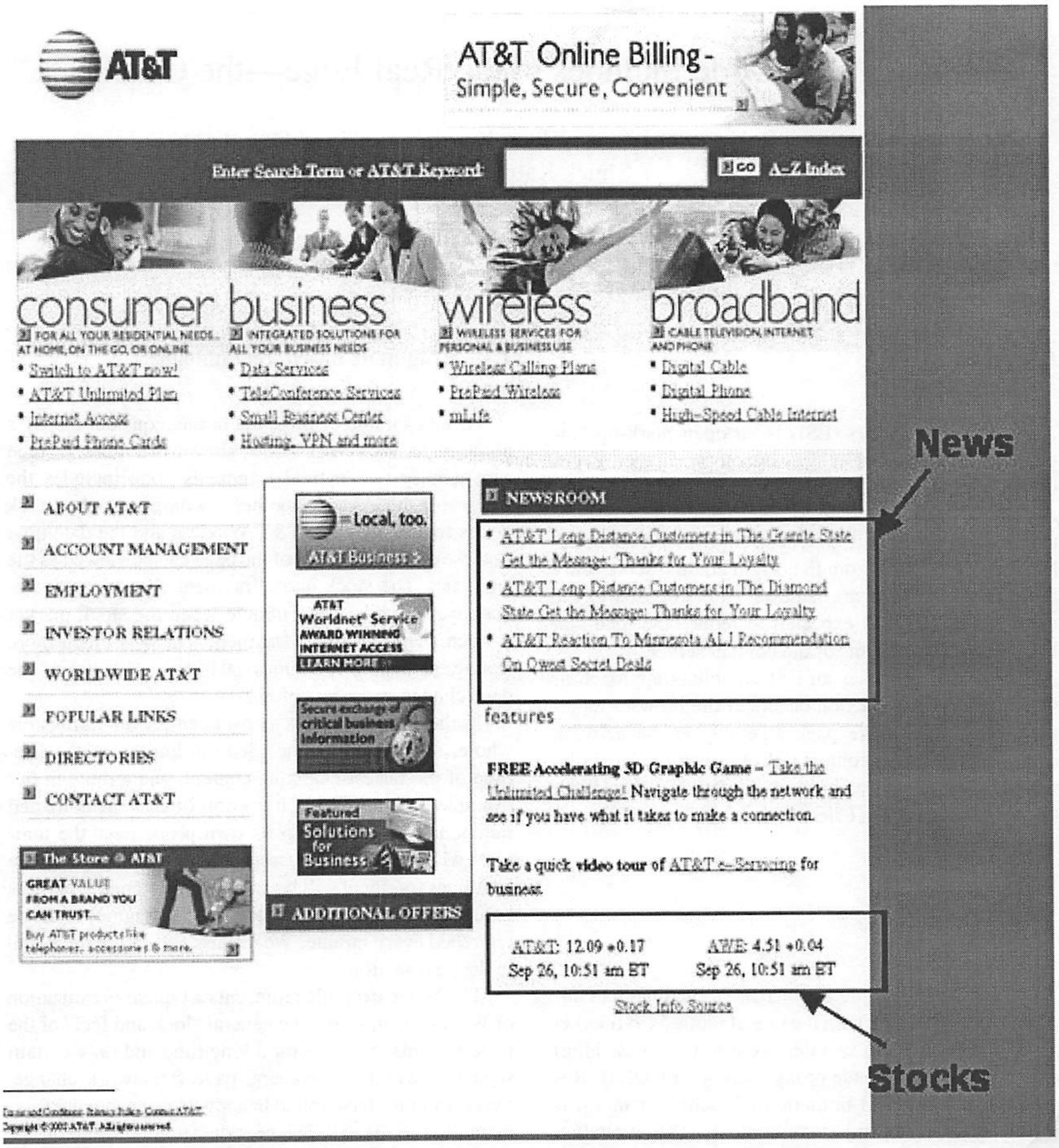
1.1 Fragment-based Technologies

As an example of fragment usage, consider AT&T's home page, `www.att.com`, shown in Figure 1. One can identify two natural fragments: one includes the newsroom headlines; another encompasses the stock prices for AT&T and AT&T Wireless and the date/time of these quotes. The rest of the page can be viewed as the template. The stock quote fragment changes very frequently, probably every minute when the stock market is open. The headlines fragments changes much more slowly, perhaps a few times each day. The rest of the page changes even less often.

If the entire page is considered as an indivisible whole, its lifetime in the cache is limited to the lifetime of the fastest-changing content, one minute in our example. With ESI, each portion of content is treated individually according to its own properties: the template will usually be accessed using the cached copy, the headlines fragment will be validated/refetched every few hours, and only the small stock quote fragment will be refetched every minute. We discuss ESI in greater detail in the next section.

The above example represents a typical organization of Web content, where the general "look and feel" of the page remains the same for a long time and only certain segments within this general page framework change. Several studies have found benefits from such page fragmentation [6, 21, 5]. The next question is where on the processing path from the origin server to the browser to reassemble a fragmented page? Historically, page assembly was first done at origin sites using technologies like Active Server Pages [1], Java Server Pages [11], PHP Hypertext Preprocessor [18], and Server-Side Includes [20]. The motivation behind this approach includes simplification of maintenance of Web sites (e.g., templates can be stored as static files and populated in a systematic way using data extracted from databases; different fragments can be conveniently generated by specialized application servers, etc).

Akamai implemented page reconstruction at the *edge*



AT&T

AT&T Online Billing - Simple, Secure, Convenient

Enter Search Term or AT&T Keyword: GO A–Z Index

consumer — FOR ALL YOUR RESIDENTIAL NEEDS... AT HOME, ON THE GO, OR ONLINE
- Switch to AT&T now!
- AT&T Unlimited Plan
- Internet Access
- PrePaid Phone Cards

business — INTEGRATED SOLUTIONS FOR ALL YOUR BUSINESS NEEDS
- Data Services
- TeleConference Services
- Small Business Center
- Hosting, VPN and more

wireless — WIRELESS SERVICES FOR PERSONAL & BUSINESS USE
- Wireless Calling Plans
- PrePaid Wireless
- mLife

broadband — CABLE TELEVISION, INTERNET AND PHONE
- Digital Cable
- Digital Phone
- High-Speed Cable Internet

ABOUT AT&T
ACCOUNT MANAGEMENT
EMPLOYMENT
INVESTOR RELATIONS
WORLDWIDE AT&T
POPULAR LINKS
DIRECTORIES
CONTACT AT&T

The Store @ AT&T — GREAT VALUE FROM A BRAND YOU CAN TRUST... Buy AT&T products like telephones, accessories & more.

= Local, too. AT&T Business >
AT&T Worldnet® Service AWARD WINNING INTERNET ACCESS LEARN MORE >>
Secure exchange of critical business information
Featured Solutions for Business
ADDITIONAL OFFERS

NEWSROOM
- AT&T Long Distance Customers in The Granite State Get the Message: Thanks for Your Loyalty
- AT&T Long Distance Customers in The Diamond State Get the Message: Thanks for Your Loyalty
- AT&T Reaction To Minnesota ALJ Recommendation On Qwest Secret Deals

features

FREE Accelerating 3D Graphic Game – Take the Unlimited Challenge! Navigate through the network and see if you have what it takes to make a connection.

Take a quick **video** tour of AT&T e-Servicing for business.

AT&T: 12.09 +0.17	AWE: 4.51 +0.04
Sep 26, 10:51 am ET	Sep 26, 10:51 am ET

Stock Info Source

Terms and Conditions Privacy Policy Contact AT&T
Copyright © 2002 AT&T. All rights reserved.

Figure 1: An example of a page amenable to ESI encoding, with two "fragments," *news* and *stocks*, highlighted by boxes.

servers, outside the origin Web site, and ultimately coauthored the ESI language specification [8]. Other vendors supporting ESI include Speedera, a CDN that offers ESI page assembly at the edge similar to Akamai, and IBM, which offers edge servers supporting this functionality.

1.2 CSI: Addressing the Last Mile

ESI was proposed with the goal of assembling the page on surrogates: reverse proxies that act on behalf of the origin server in serving client requests, or edge servers in the CDN parlance. The ESI overview [10] claims that ESI "speeds up delivery of highly dynamic Web-based applications," in addition to the other goals of reductions in network and server loads. However, in this paper, we observe that page assembly at the edge server does not improve the response time for dial-up clients, which still represent a large majority of Web users (79% of consumer subscribers as of March 2002 [17]) and, while declining, are projected to remain a majority for the next several years (59% of all on-line households in the US in 2006 [12]). The reason for the lack of improvement in the dial-up environment is that the dial-up link (referred to as "the last mile") is often the bottleneck that determines the download time, and edge assembly does not affect the amount of content over that link. Furthermore, depending on the nature of content, ESI may actually increase the load on origin servers (see Section 4).

Consequently, we implemented an alternative mechanism, which performs ESI page assembly directly in the browser. We found that assembly in the browser can dramatically reduce the user response time. Depending on the nature of the page, we observed a significant reduction in the page display time for dial-up users with 56Kbps modems. Because page assembly occurs on the client, we call our approach *Client-Side Includes*, or *CSI*.

Our mechanism requires no modification of browsers or configuration of the browser machine. In particular, it does not require such typical extension techniques as configuring a new browser plug-in, or co-locating a custom proxy at the browser machine. At the same time, our mechanism implements most of the language specification of ESI 1.0 and thus enables content providers using ESI to switch to our mechanism without changing their content.[1]

An important advantage of CSI is that, unlike edge-based page assembly, CSI does not require the presence of an edge server. CSI can be implemented between the origin server and the browser directly. The existence of the edge server, and indeed the use of a CDN by the Web site, becomes an orthogonal issue. When a CDN is used, CSI can utilize edge servers for scalable delivery of page templates and fragments. Otherwise, browsers can download them directly from the origin server. This flexibility is important because inserting and removing ESI mark-up, however simple it might appear, has a high administrative overhead for large Web sites. With edge-side page assembly, a decision to use ESI entails a commitment for continued use of a CDN. With CSI, a Web site is free to use or not use a CDN based on other factors, such as performance and price.

Moreover, when the content provider does use a CDN, CSI can significantly reduce their CDN-related costs. The reason is that CDNs charge content providers for the traffic they deliver from their edge servers to clients. CSI reduces this traffic by delivering only ESI fragments, and not entire pages, from edge servers to clients.

It is important to realize the difference between a mark-up language and the mechanism for page assembly. Both our proposed CSI and the existing edge-side assembly mechanisms use the same mark-up language, called ESI. We will be careful to distinguish between the ESI *language* and the ESI *assembly mechanism* unless the meaning of the "ESI" term is clear from the context.

1.3 Our Contributions

While client-side assembly of a page from individual components has been described before [6, 3], our paper makes a number of contributions beyond prior work:

- While existing work used their own ad-hoc fragmentation languages, ours is to our knowledge the first paper that implements client-side page assembly using an existing widely supported language for page fragmentation, which was independently proposed for a different purpose. This is important because legacy content that already uses this language can benefit from our approach, whereas previous work required re-authoring the content.
- A significant concern with client-side implementations is the need to support the implementation across a vast number of different browser types and versions that co-exist on the Internet. We demonstrate that this concern can be effectively addressed without modifying the server code and without the need to maintain multiple versions of the content.
- To our knowledge, ours is the first paper that raises and addresses an issue of assessing whether and which pages should use fragmentation on a Web site.
- Finally, we provide details of our prototype implementation. While obviously dependent on the cur-

[1] Our current implementation excludes support for the optional "inlining" feature and provides incomplete support for ESI variables.

rent technology context, the lessons from our implementation experience will be useful for readers who implement other functionality on the clients.

2 ESI Overview

ESI is an open-standard XML-based markup language that provides a mechanism to assemble a web page from different components at the edge of a network. Each component can be retrieved independently and have its own cache control header, such as an expiration time. Since many web pages have a mixture of dynamic content (like stock quotes) and static content (like a page template), this design facilitates caching of web objects. Ideally, it helps reduce the congestion on the network, reduces the processing overhead on origin servers, and improves overall response time [10]. ESI has been developed by a consortium and subsequently proposed as a standard within the World Wide Web Consortium [8]. It is primarily targeted at processing on surrogates: reverse proxies that act on behalf of the origin server in serving client requests.

```
1. <HTML>
2. <!--esi
3.   <H3>Stock quote for ${QUERY_STRING}</H3>
4.   <esi:try>
5    <esi:attempt>
6.     <esi:include src=/quote.html
7.               alt=/delayed_quote.html/>
8.     <esi:choose>
9.       <esi:when
10.        test="$(HTTP_COOKIE{Type})==premium">
11.        <esi:include src=/market_news.html/>
12.      </esi:when>
13.      <esi:otherwise>
14.      To subscribe to premium services
15.        <A href=/subscribe.html> click here </A>
16.      </esi:otherwise>
17.    </esi:choose>
18.   </esi:attempt>
19.   <esi:except>
20.     <esi:include src=/sorry.html />
21.   </esi:except>
22. </esi:try>
23. -->
24. <esi:remove>
25.   Please click on a
26.   <A href=/non_esi.html> non-ESI version </A> of this site
27. </esi:remove>
</HTML>
```

Figure 2: An example of ESI usage.

Figure 2 shows an example of an ESI template. The `<!--esi` and `esi:remove` tags allow templates that can be handled by non-ESI reverse proxies and browsers. If our template is obtained by a browser directly or through a non-ESI reverse proxy, the browser will assume that line 2 signifies the beginning of HTML comments and will ignore lines 3-21. The browser will further ignore unknown `esi:remove` tags and will display an invitation to access a non-ESI version of the site on lines 25-26. An ESI processor, conversely, removes `<!--esi` tags from the final page, and treats the text within the `esi:remove` block as comments. Thus, lines 3-21 will be processed and the invitation to access the non-ESI version will be elided.

Turning now to processing of lines 3-21, the `attempt` block of lines 5-18 is executed first. Line 6 tries to insert a fragment with relative URL "/quote.html". If for some reason this download fails (e.g., the data feed was broken), a fragment "delayed_quote.html" is tried next. If this download, or any other download in the attempt block, fails, the `except` block on lines 19-21 is executed.

Lines 8-17 give an example of a conditional inclusion based on testing a request cookie. Presumably the request Cookie header could contain "User-Type=premium" field, in which case line 11 would include the market news fragment, otherwise the invitation to subscribe to this service would be inserted into the final page. The `esi:remove` tag instructs the ESI processor to remove the enclosed text (the invitation to access non-ESI version of the site in our example) from the final page. The client that does not understand ESI would ignore the unknown tags and will display the text on lines 25-26.

This example illustrates fragment inclusion, conditional inclusion, and exception handling of ESI. Other features include a possibility of nested ESI constructs and access to environment variables. In particular, inclusion in ESI can be nested: the "/market_news.html" fragment in our example can itself include other fragments (or other ESI mark-up). Later versions of ESI also allow the template to declare and assign values to arbitrary variables, which can then be accessed (e.g., tested for conditional inclusion or inserted the assembled HTML page) inside ESI fragments.

3 Our Approach

There are a number of ways to assemble a page from various segments of information. The most common way is to do it at the origin server as shown in Figure 3a. With ESI, Akamai and others moved page assembly to CDNs' edge servers (Figure 3b). By reusing unchanged content from edge servers' caches, this approach reduces

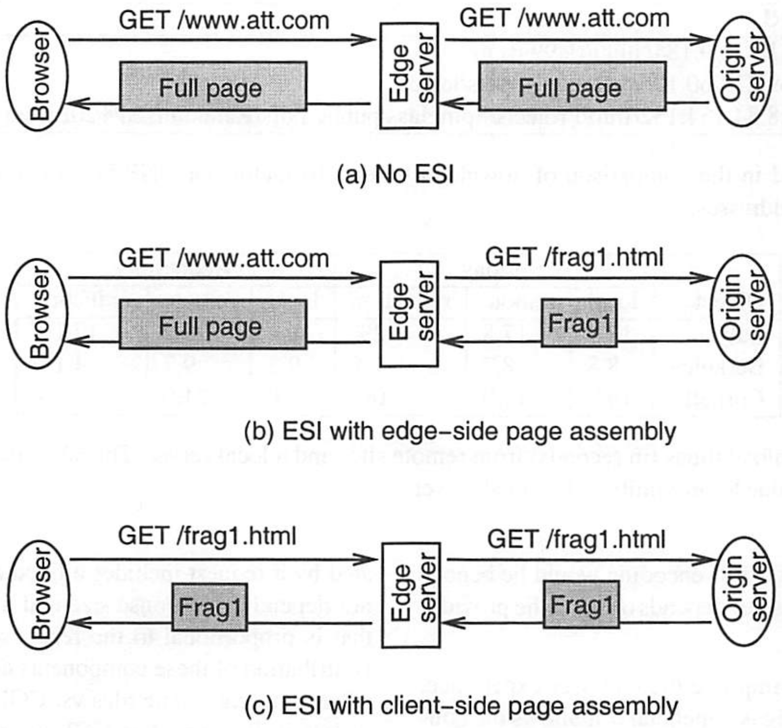


Figure 3: Page assembly alternatives assuming only fragment Frag1 must be prefetched into the cache.

traffic outflow from origin sites and associated connectivity costs for content providers. Also, depending on the nature of content, edge assembly may reduce server load (although it may also have an opposite effect as discussed in Section 4). By reducing bandwidth consumption and possibly server load, edge assembly can improve the peak capacity of origin sites. However, intuitively, edge assembly would not reduce download times for dial-up users during normal operation because the determining factor in this case is the slow dial-up link.

A separate study would be needed to conclusively verify this intuition. As a preliminary indication, we used a *wget* tool to download objects of around 100K from some remote sites (listed in Table 1) to a local idle server and compared the time it takes a dial-up client to fetch these objects from the remote sites and the local server. The client used a 56Kbps modem dialing into a modem bank that shared a building and a LAN with the local server. Thus, downloading from the local server emulated the best possible scenario of CDN-facilitated content delivery. As Table 2 shows, there is no discernible difference in median download times from remote and local servers except for the Wednesday experiment with Italy, which had two extremely high download times, 44 and 65 seconds.[2] We speculate that these two downloads reflect dropped connections by origin servers.

We propose to push page assembly further, to the ultimate network edge – the client itself. In our *CSI* approach, illustrated in Figure 3c, the client downloads only changed fragments, thereby reducing both the traffic that flows out of the origin server and that is transmitted over the last-mile link connecting the client to the Internet.

Just like edge assembly, CSI reduces connectivity costs for the content provider and possibly improves the origin site peak capacity. But it also improves the user experience for those with dial-up or other low-bandwidth connections. Furthermore, as already discussed in the Introduction, unlike edge assembly, CSI is applicable to Web sites regardless of whether or not they use CDN servers, and it reduces CDN-related costs for those Web sites that do use CDNs.

4 ESI or not ESI?

Before we explore different alternatives for ESI page assembly, an important question is how a content

[2]The large difference in download times between the three objects reflects the different degree of compressibility, since modems use hardware compression. As an indication of compressibility, the object size after *gzip* compression remained almost unchanged for the Italian object, reduced slightly to over 99K for the Cornell object, and reduced by more than the factor of 3, to just over 30K, for the Berkeley object.

Location	URL	Size
Italy	131.114.9.184/ luigi/rlc99.ps.gz	96983
Berkeley	169.229.60.105/ helenjw/papers/icc.ps	96176
Cornell	128.84.154.132/Info/Projects/Spinglass/public_pdfs/Randomized%20Error.pdf	115788

Table 1: Objects used in the comparison of download times. To factor out DNS latency, host names have been pre-resolved into IP addresses.

	Friday			Wednesday		
Object	local	remote	reduction	local	remote	reduction
Italy	17.5	17.8	2%	18.4	22.2	17.1%
Berkeley	8.8	8.7	-1%	9.3	9.7	4.1%
Cornell	19.9	19.9	0%	21.0	21.0	0%

Table 2: Median download times (in seconds) from remote sites and a local server. The table also shows the percentage of improvements due to proximity of the local server.

provider could decide if ESI encoding would be beneficial in its case. The answer depends on what the provider is trying to optimize.

- If the goal is to improve the end user experience, then ESI encoding is beneficial if it allows the content provider to move a substantial portion of content to templates or fragments with long TTLs. For poorly connected clients, the benefits are greatest if CSI assembly is used rather than ESI assembly, since traffic is reduced over the bottleneck link.
- If the goal is to reduce bandwidth consumption on the link from the Web server to the network, the condition that a substantial portion of content belong to ESI objects with long TTL will ensure that ESI is beneficial, regardless of the assembly method.
- Finally, if the goal is to increase the effective server capacity, the answer depends on the nature of the content. The following example elaborates upon this issue.

As an extreme example of a situation where ESI would be detrimental to server capacity, consider a 20K page that has three mutable regions of 2K each that change every minute. According to the ESI literature, this page seems like a good candidate for ESI encoding. However, for every HTTP request that reached the server without ESI, there will be either four requests (if it is a brand new client) or three (if is it a repeat client with a cached template). Overall, in the best-case scenario for ESI, when all requests are due to repeat clients, the bandwidth consumption out of the server reduces by a factor of three ($20K/(3 * 2K) = 3.3$), but the number of requests to the server increases three-fold.

If the bottleneck is the server load, the overall effect can easily be detrimental. Indeed, the server load generated by a request includes a fixed component that does not depend on response size and a variable component that is proportional to the response size. The relative contribution of these components depends on the nature of content (e.g., static files vs. CGI scripts in C vs. CGI scripts in Perl vs. fast CGI or servlets), and whether or not persistent connections are used. As one indication, we tested the request throughput of Apache 2.0.40 on a 733MHz NetBSD machine and, using static files with persistent connections, the throughput for a 20K page was reduced by only a factor of 2 as compared with the throughput for a 2K page – 1150 vs. 2274 requests per second.[3] So, in our example, with this workload, ESI would prove detrimental for effective server capacity even though it reduces the number of bytes served.

4.1 Guidelines

To generalize the above example, we propose the following procedure to infer the overall effect of reduced bandwidth consumption and increased request rate on the origin server capacity. We will use Figure 4 to illustrate the procedure, and we will explain various elements in this figure as we move along.

- Stress-test the server to see how its request throughput depends on the size of the response and plot the result. For example, the descending solid line on Figure 4 shows the throughput vs. response size curve obtained in our experiment with static files mentioned earlier. Call this a *capacity curve*. In obtaining the capacity curve, use resource types that approximate resources to be used in practice (e.g., Figure 4 was obtained using static files, which would be appropriate for sites like att.com, where

[3]A very similar trend can be seen in results by Nahum et al. for a variety of Web servers – see Table II in [16].

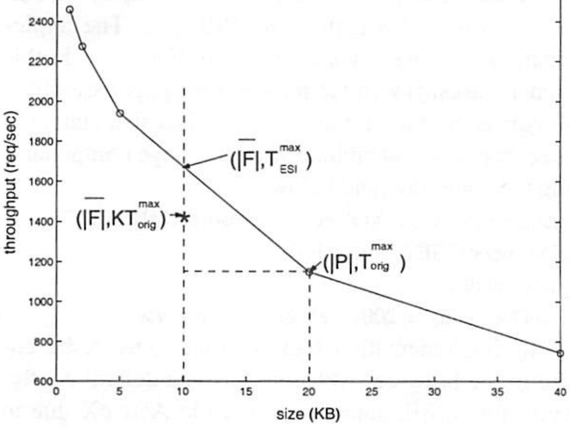


Figure 4: Effect of ESI encoding on server load.

even the most mutable fragments are only updated periodically; for other resources, servlets or CGI scripts might be more appropriate).

- Let $|P|$ be the size of the original page in question, that is, the page for which we must decide whether to use ESI encoding. Estimate (we will discuss how later) the average size of ESI objects, $|\bar{F}|$, that would be shipped from the server if the page were ESI-encoded and by how many times the request rate would increase, K. That is, if T_{orig} is the request rate for the original page, KT_{orig} will be the total request rate for the page template and fragments after it is ESI-encoded. Note that $|\bar{F}|$ is the average over server responses rather than the number of fragments on the page, so a more frequently requested fragment contributes more to the average.

- Draw two vertical lines in the throughput vs. object size plane, corresponding to the original and average ESI-encoded response size, $|P|$ and $|\bar{F}|$. For instance, the two dashed vertical lines in Figure 4 correspond to the case where the original page size is 20K and the average ESI-encoded response size will be 10K. Let $(|P|, T_{orig}^{max})$ and $|(\bar{F}|, T_{ESI}^{max})$ be the points where these vertical lines intersect with the capacity curve. T_{orig}^{max} represents the server capacity for the original page and T_{ESI}^{max} gives the server capacity for the ESI-encoded page.

- We know that, whatever the request rate for the original page was, the request rate for the ESI objects after ESI encoding will be K times higher. In particular, the maximum sustainable demand for the original page (corresponding to T_{orig}^{max} request rate) results in the request rate for the ESI objects that is equal to $T_{after} = K \times T_{orig}^{max}$. If the request rate $T_{after} = T_{ESI}^{max}$, then the server will remain fully utilized after ESI-encoding, and its effective capacity will remain the same. If $T_{after} < T_{ESI}^{max}$, the server utilization will be below capacity and hence it could serve some additional requests. In other words, ESI encoding would increase the effective capacity of the server. Finally, if $T_{after} > T_{ESI}^{max}$, ESI encoding will reduce the effective capacity. Graphically, we plot the point $(|(\bar{F}|, K \times T_{orig}^{max})$ on the coordinate plane. If this point is above the throughput curve, ESI encoding of page P will reduce the effective server capacity, otherwise the capacity will increase. For example, in Figure 4, T_{orig}^{max} is 1150 requests per second. Assuming $K = 1.25$, the same demand that drove the server to capacity would result in $1150 \times 1.25 = 1437.5$ requests per second for ESI objects, which falls below the capacity curve and thus indicates that ESI encoding of this page would be beneficial from the perspective of server capacity.

4.2 Estimating $|\bar{F}|$ and K

To estimate $|\bar{F}|$ and K, the most general technique is a trace simulation using the Web server access log. The simulator should translate each request for page P in the log into requests for the template and every fragment, and then filter out repeated requests from the same client that would hit in the client's cache, taking into account TTLs of individual ESI objects. The remaining requests could then be used to calculate $|\bar{F}|$ and the new request rate.

However, some special cases, such as the case where every fragment has a non-zero TTL, allow analytical estimations. Consider a Web site that uses a CDN and a page P of size $|P|$ that contains n fragments, $F_1, ..., F_n$, with each fragment F_i having a lifetime of t_i seconds and size $|F_i|$. Let t_m be the smallest lifetime of all fragments, with t_m greater than 0.

With ESI, in the best case of all repeat clients, the template is (almost) never sent. Assume that page P is popular enough so that the request rate for it at each of the CDN's edge servers is much higher than fragment lifetimes. (After all, unpopular pages do not matter from the perspective of load.) Then, without ESI, an edge server will send $1/t_m$ requests per second. With ESI, the edge server will send $\sum_{i=1}^{n}(1/t_i)$ requests per second.

Since the same is true for all edge servers, the request rate is increased by a factor of

$$K = [\sum_{i=1}^{n}(1/t_i)]t_m.$$

```
<HTML>
  <BODY>
    <SCRIPT SRC="csi.js"> </SCRIPT>
    <SCRIPT> run("page_template.html"); </SCRIPT>
  </BODY>
</HTML>
```

Figure 6: The wrapper for Javascript/ActiveX implementations of CSI.

Similarly, the amount of data served per second to the edge server is $\sum_{i=1}^{n}(|F_i|/t_i)$ and therefore the average response size is

$$|\bar{F}| = \frac{\sum_{i=1}^{n}(|F_i|/t_i)}{\sum_{i=1}^{n}(1/t_i)}.$$

5 Implementation

CSI is a mechanism for assembling a page from individual ESI components at the browser. Any implementation of CSI must be able to download page components, process them to assemble the page, and tell the browser to display the result. We implemented CSI using JavaScript for assembling the page. Our implementation follows the framework shown in Figure 5. When a CSI-capable client requests a page, the server returns a small wrapper (150 bytes plus headers). The wrapper invokes a Javascript page assembler and passes it the URL of the ESI template corresponding to the requested page. The page assembler then downloads the template and any ESI fragments it includes and assembles the page.

It might appear that CSI involves much overhead to download the page assembler script and the page wrapper. However, the assembler script is generic for all CSI content from any Web site. Thus, once a client downloads it, it remains in its cache and is invoked locally. This is akin to installing a piece of software on the client except this software is installed transparently the first time it is used. The wrapper is immutable for a given page but not across pages because it includes the URL of the requested page. Thus, the client does incur an overhead of fetching the wrapper when it accesses the page for the first time. Subsequent accesses to this page will not incur this overhead because they will reuse the cached wrapper even if the page itself has changed. Because the wrapper is very small, the above overhead is mostly due to round-trip packet latency and not bandwidth consumption.

We implemented CSI for Microsoft's Internet Explorer browser using ActiveX to download page components. In principle, Java's LiveConnect facility can be used instead, except we in the past encountered incomplete support of this feature in MSIE [7]. The implementation uses the wrapper shown in Figure 6. In this wrapper, the csi.js script implements the page assembler, and `run` is the function in the assembler that starts the processing. The assembler downloads page components using the code fragment below:

```
httpDoc = new ActiveXObject("Microsoft.XMLHTTP");
httpDoc.open("GET", url, false);
httpDoc.send();
if (httpDoc.status != 200) <Process exception>
```

This implementation requires that ActiveX be enabled in the browser. Although this is a default configuration for MSIE, some users disable ActiveX due to security concerns, in which case we resort to a fall back approach that we also use for non-IE browsers. We discuss this fall-back approach next.

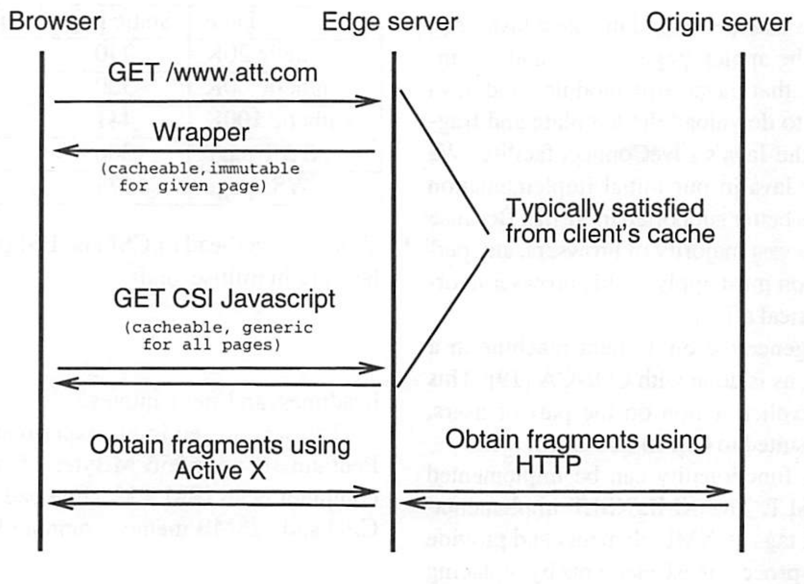
5.1 Supporting Non-IE Browsers

Our implementation of CSI only works for Microsoft Internet Explorer (MSIE). Although MSIE now occupies the overwhelming majority of the browser market, an important issue we need to address is how to deal with non-MSIE browsers and various early versions of the MSIE browsers that might not be compatible with the Javascript or ActiveX features used by our implementation. One could attempt to support different CSI implementations for all possible browsers. This would be an administrative nightmare and a significant disadvantage over edge assembly of ESI pages which is browser-agnostic. Alternatively, a Web site could maintain two versions of the content, one with ESI markups for CSI-capable clients and the other for other clients. However, maintaining the content in two versions is unacceptable to many content providers due to administrative costs.

Our approach to this problem is to implement CSI for the prevalent browser only (recent versions of MSIE) and to resort to edge-side or server-side page assembly for all other browsers. This approach can be implemented in two ways, which decide between CSI and ESI either at the client or at the server.

The client-side solution adds a test for browser capabilities to the wrapper and redirects the non-CSI capable browser to the ESI script with the template URL as a parameter. Figure 7 shows an example of such wrapper for the ActiveX CSI implementation. If the browser does not support Javascript, the wrapper invites the user to click for the ESI script. Although awkward, this seems acceptable as Javascript is virtually universally enabled. The disadvantage of the client-side solution is that it increases the size of the wrapper.

The server-side solution is enabled by HTTP's User-agent header, which nearly all browsers include with



Browser
Edge server
Origin server
GET /www.att.com
Wrapper
(cacheable, immutable for given page)
Typically satisfied from client's cache
GET CSI Javascript
(cacheable, generic for all pages)
Obtain fragments using Active X
Obtain fragments using HTTP

Figure 5: CSI interactions for a browser that never before accessed any CSI-enabled page.

```
<HTML>
  <BODY>
    <SCRIPT>
    <! – –
      if (!window.ActiveXObject)
        window.location="/cgi-bin/esi.pl/template.html"
    //-->
    </SCRIPT>
    <SCRIPT SRC="csi.js"> </SCRIPT>
    <SCRIPT>
    <!--
      run("template.xml");
    //-->
    </SCRIPT>
    If your browser does not support Javascript please click
    <A href="/cgi-bin/esi.pl/template.html"> here </A>
  </BODY>
</HTML>
```

Figure 7: The wrapper choosing between client- and server-side page assembly.

their requests. If the server processing the request (the origin server or edge server if CDN is used) can determine from this header that the client is CSI-capable, the server returns the CSI wrapper. Otherwise, the server reconstructs the page itself and returns the complete HTML page. This client-specific request processing can be done without any modification of server code. On Apache, it can be achieved by an appropriate server configuration. One configuration method, which we tested, uses Apache's URL rewriting [14]. Here, a requested URL is rewritten into different internal URLs depending on the value of the User-agent field in the HTTP request header. The disadvantage of the server-side solution is that it does not work well with client Web proxies. The server now returns different responses for the same request based on the nature of the browser. If a proxy has both CSI-capable and CSI-incapable clients, the proxy should also distinguish between these responses and not send a CSI response to a CSI-incapable browser. The server can ensure this behavior by adding "Vary: User-agent" HTTP header to its responses. Unfortunately, some proxies do not cache responses with this header, while those that do will not share a cached response among *any* non-identical browsers even if all of them support CSI (such as different versions of MSIE). In either case, the effectiveness of proxy caching would be reduced.

5.2 Discussion

There are a number of other ways to implement the CSI functionality. Possible approaches include using pure Java, Javascript/Java combinations; pure Javascript; local proxy caches; and XML with XSLT transformations.

In the pure Java approach, the exposed URL of an ESI-encoded page would return a small wrapper object, which would invoke the applet that implements CSI, passing to this applet the URL of the template. The applet would fetch the template and assemble the page, downloading page fragments as needed, and then display the page. The applet itself would be generic for any ESI-encoded page and would typically be found in the browser cache. The Javascript/Java combination would be similar, except the wrapper would invoke a Javascript module rather than the applet, page parsing and assembly would occur in that Javascript module, and Java would only be used to download the template and fragments by means of the Java's LiveConnect facility. We chose ActiveX over Java in our initial implementation because the former is better supported in MSIE. Because MSIE constitutes the vast majority of browsers, any performance optimization must apply to this browser in order to have any practical effect.

Pages could be generated on a client machine in a separate application, as is done with CONCA [19]. This approach requires explicit action on the part of users, and is therefore not suited to CSI in its current form.

A subset of ESI functionality can be implemented using XML and XSLT. The XML/XSLT implementation would treat ESI tags as XML elements and provide XSLT procedures to process these elements by replacing them with appropriate ESI fragments. The advantages of this approach is that the XML/XSLT implementation does not require a separate wrapper and that XML/XSLT is being adopted across most browsers. However, its disadvantage is that it does not have access to HTTP-related variables such as request header fields. As a result, it can only implement a subset of the language.

6 Performance

In this section, we measure the performance of ESI/CSI. The measurements were conducted using a set of synthetic pages and two real pages. The synthetic pages were HTML files of specified sizes with randomly generated characters. When ESI encoding is used, these pages are split into a template and a specified number of fragments. This allows us to study performance trends of ESI/CSI in a systematic manner by varying the degree of caching for the CSI JavaScript, the template, and the fragments. In this experiment, we generated synthetic pages of 20K, 60K, and 100K bytes. Each page consists of a template and four fragments. The template has 80% of bytes in the page, and each fragment has 5%. As a target for comparison, we also generated a static page for each page size by assembling the template and its fragments offline.

For the real pages, we chose AT&T's entry page, `http://www.att.com`, and the Wall Street Journal's entry page, `http://online.wsj.com/`. We downloaded a copy of each page and its embedded objects to a local Apache server and then manually split the page into an ESI template and a set of fragments. The AT&T page has two fragments (as shown in Figure 1): news headlines and stock quotes. The Wall Street Journal page has three fragments: time of the day, news headlines, and stock quotes.

Page	Static page	with CSI	with ESI
synthetic 20K	240	320	380
synthetic 60K	300	431	441
synthetic 100K	441	491	501
AT&T page	306	351	430
WSJ page	571	676	831

Table 3: Overhead of CSI and ESI processing. All numbers are in milliseconds.

The server used in the experiments runs on a 864Hz Pentium III with 256 MBytes of memory. The client computer is an IBM T22 Thinkpad laptop with a 1GHz CPU and 128MB memory running Windows 2000.

6.1 Overhead

We first measure the overhead of ESI and CSI processing. To do so, we compare the display time of an ESI-encoded page using server-side assembly or client-side assembly with that of the corresponding static page. We implemented server-side assembly as a Perl script with FastCGI. Since our software resorts to server-side assembly for clients considered CSI-incapable, we want to make sure the overhead is acceptable. The experiment was conducted over an 100Mbps Ethernet and was repeated 30 times. In repeat downloads, all page components were fetched from the browser cache. Table 3 shows the median display time for different synthetic pages as well as for the two real pages. All numbers are in milliseconds.

As can be seen from the table, the overhead for server-side assembly is small: below 300ms in all cases. This overhead can be significantly reduced by reimplementing page assembly in C. However, given that only very few clients will experience it, we consider it to be already acceptable. We should also note that in reality these pages would not be downloaded from static files anyway. Rather, they need to be assembled on the server using some mechanism such as server-side includes. Thus, our measurements provide an upper bound of the overhead.

The table also indicates that the overhead for CSI assembly is under 150ms. As we will see in the next subsection, this overhead is more than offset by savings in transfer times for dial-up clients. While for broadband clients it might result in a slight increase of the overall download time, we believe it is justified by a significant reduction in download time for dial-up clients.

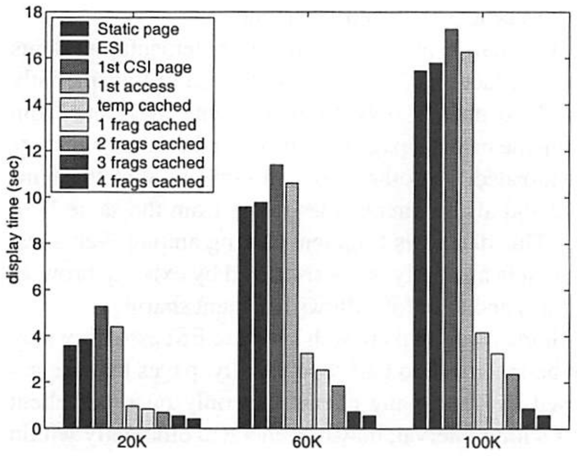


Figure 8: Download time of synthetic pages over dial-up links.

6.2 Display Time

Next we measure the time it takes for a client to retrieve a page from the Web server and display it in its browser over dial-up links with 56K modems. The results for synthetic pages are shown in Figure 8. The figure indicates that the display time for ESI is slightly higher than that of the static page due to the processing overhead of the ESI Perl script. The figure also shows that there is a substantial penalty in performance when a client invokes CSI for the first time (denoted as "1st CSI page" in the figure). In this case, the browser needs to download the CSI JavaScript, the wrapper page, the template, and all the fragments. For 20K pages, it increases the display time by as much as 48%. Since the CSI script is the same for all pages, it can be served from the browser cache afterward. Even so, the first access to an CSI-enabled page (denoted as "1st access") may still incur an overhead due to the download of the wrapper page and the processing of CSI script inside the client's browser. However, during subsequent visits to the page, the template is likely to be served out of the cache since it seldom changes. Consequently, only those fragments that have changed need to be fetched from the server. As can be seen from the figure, this results in substantial reduction in display times across all page sizes.

The results for the AT&T page are shown in Figure 9. The figure indicates that CSI improved the median download time by about 25%, from 3450ms to 2569ms, assuming that the template is cached, which would be the typical case. Note that the performance improvement for the AT&T page is not as substantial as that for synthetic pages. We discovered that this is because the AT&T page is more compressible than randomly gener-

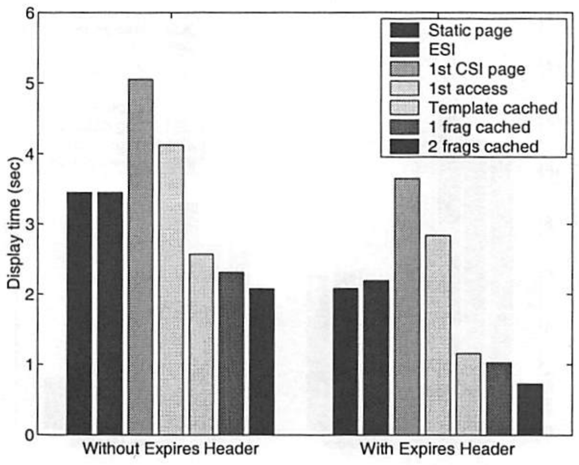


Figure 9: Download time of the AT&T entry page over dial-up links.

ated synthetic pages.

We further observed that the AT&T site (like many other Web sites) does not specify expiration times for its embedded objects. MSIE in this case sends "If-Modified-Since" requests to validate these objects on each access. Since these objects (mostly images) do not change often, we configured the local server to provide expiration times for them. Then we repeated the experiments to measure the download time for what could be considered a "properly configured" Web site. With explicit expiration times, all values reduced accordingly and CSI's relative improvement grew to 45%. Like in synthetic pages, the first access to the AT&T page may have a high overhead due to the download of the wrapper page, the template, and all the fragments.

Figure 10 shows the results for the Wall Street Journal page. When the template was served from the browser cache, CSI reduces the display time by about 27%. If expiration headers were provided for the embeded objects, the improvement becomes 38%. Note that some of these objects have query strings in their URLs. This causes MSIE to send validation requests even if they have not expired.

6.3 Bandwidth Reduction

An important goal of our project is to reduce the amount of bytes that need to be transmitted over the last mile. Table 4 shows the sizes of the two real pages and their components with ESI-encoding.[4] The table indicates that 93% of the bytes in the AT&T page and 71% of the bytes in the Wall Street Journal page are in their

[4]The total size of the page template plus all its fragments is slightly higher than the size of the full page. This is due to some ESI encoding statements we added into the page.

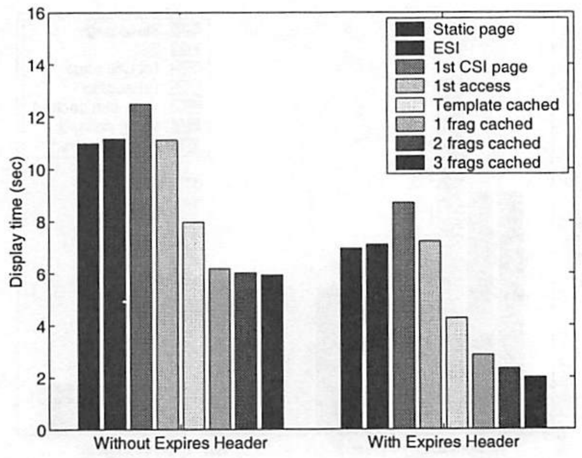


Figure 10: Download time of the Wall Street Journal entry page over dial-up links.

	AT&T Page	WSJ Page
full page	30731 (100%)	79608 (100%)
page template	28661 (93%)	56324 (71%)
current time	N/A	55 (0%)
news headlines	927 (3%)	20161 (25%)
stock quotes	1231 (4%)	3166 (4%)

Table 4: Sizes of the AT&T page and the Wall Street Journal page with ESI encoding.

templates. Hence, client-side assembly of these pages can achieve significant reduction in bandwidth when the templates are in the browser's cache. Note that the actual saving observed in practice is slightly lower than indicated in the table because of the extra bandwidth consumed by HTTP requests for the page components and by HTTP headers carried by responses with the components. We estimate that this adds about 400 bytes for each fragment that is not in the cache.

7 Limitations and Future Work

The need to download the wrapper increases latency when the browser accesses the page for the first time and hence does not have the wrapper in its cache. Our particular implementation downloads fragments into the CSI assembler script sequentially and synchronously with template parsing. This may slow down page assembly for pages containing a large number of fragments that are not locally cached. Furthermore, MSIE seems to always validate any locally cached object that does not have an explicit expiration time, and in our case these fragment validations would be sequential. Thus, CSI will bring higher benefits to those Web sites that supply Expires headers for their fragments.

A limitation not specific to our implementation stems from the fact that Javascript is allowed to transparently download objects only from the same Web site from which the original page (that invoked the Javascript) was downloaded. In other case, this means that the template and all fragments must come from the same Web site. This disallows fragment sharing among Web sites. Edge-side assembly is not restricted by existing browser realities and therefore allows fragment sharing.

Some pages that are well suited to ESI assembly may not be amenable to CSI. Specifically, pages that are accessed by very many clients, but only once per client over a long interval, may be generated efficiently within a CDN but slow down individual CSI clients due to the "first access" downloads discussed in the previous section.

As future work, we would like to extend the ESI language with some features from the HPP markup language [6], such as the loop construct. This would expand the applicability of ESI to more content, most notably responses from search engines.

8 Related Work

Numerous language constructs exist for including fragments into an HTML page. IMG and APPLET HTML tags tell the browser to insert, respectively, an image or an applet to the HTML document. The OBJECT tag allows an insertion of an object of an arbitrary type. These tags, however, only allow a straightforward inclusion. In particular, they allow no conditional inclusion, no access to environment variables and HTTP headers, and no user-defined variables to pass data from the containing page to included objects. The same is true of the Xinclude [22] construct in XML and XML with XSLT transformation. The ESI language allows much greater flexibility in specifying how the final page should be assembled.

Much more sophisticated inclusion mechanisms than the above-mentioned tags exist for use on the server-side. ASP, JSP, PHP, and Server-Side Includes are pure server-side tools. Unlike CSI, the page using these language constructs is assembled at the server and shipped to the client in its entirety. Thus, the client only sees the final HTML page without any inclusion constructs. The primary goal of these tools is to simplify the development and management of the Web site (for example, to provide a systematic way of organizing a database-driven Web site), and not to improve the performance of accessing the site. The purpose of the ESI language, and the CSI assembly mechanism in particular, is to improve performance.

The closest approach to CSI are HTML Pre-Processing (HPP) [6] and the <bigwig> project [3]. HPP is geared towards the case where a page contains access-specific information that changes on every access. In particular, HPP distinguishes only two kinds of content on the page - the static template and (possibly disjoint) dynamic portions that must always be downloaded from the server. The ESI language allows the containing page to specify multiple fragments, each with its own caching characteristics. At the same time, some language features from HPP, most notably its loop construct, would benefit the ESI language. In terms of page reassembly, HPP was implemented as a browser plug-in, thus requiring browser configuration. In contrast, CSI requires no such configuration. <bigwig> is similar to CSI in that it uses Javascript for page reconstruction on the client. However, it is based on its own language for Web service specification, and is applicable only to applications created with that language.

Active Cache [4] and CONCA [19] are two examples of dynamic content generation within intermediaries such as proxy caches. Unlike ESI page assembly, which uses surrogates associated with the content provider, these systems permit dynamic content to be generated closer to end users, and possibly under their control. In particular, CONCA permits user profiles to assist with transcoding content formats to a user's specifications. With CSI, our emphasis is to generate content dynamically with an established language (the ESI language) and existing browser technologies, Javascript and ActiveX.

A recently emerged notion of "utility computing" aims at providing an even higher degree of flexibility in page construction. ACDN [13] and Vmatrix [2] are examples of research efforts, and Ejasent [9] is one start-up commercial venture in this area. Rather than providing a mark-up language to insert dynamic fragments, the idea of utility computing is to allow entire applications to run at CDN servers and to let these applications migrate or be replicated among CDN servers as needed by changing demand. Utility computing is a pure edge-side approach. It aims at replicating applications rather than optimizing data transfer over the last mile. Thus, while this technology does overlap with edge-side page assembly, it is complimentary to CSI.

Finally, like previous systems like HPP and Conca, the benefits of CSI should be evaluated relative to another emerging technology, delta-encoding [15]. Sending updates to web pages, rather than the pages themselves, can save bandwidth and improve response time; it can be deployed over the last mile, between the user and a proxy; between a proxy (including a CDN) and a content provider; or across the entire connection. For the last mile, CSI has the advantage of transparency, since there need be no special support in the browser or a local proxy. It also does not require synchronization between clients and servers to specify base versions against which to compute deltas, and does not require the same content to be transmitted for different pages [19].

9 Conclusions

Numerous methods for template-based page construction exist. These methods are typically oriented toward simplifying content generation and management are intended to be executed at the origin site before the page is shipped out. The ESI language was proposed with the goal of shifting template-based page assembly to the network edge. This paper argues for shifting such page assembly all the way to the browser and shows that such a shift can occur transparently to browsers, while supporting most of the ESI language.

Our proposed client-side page assembly (CSI) not only reduces the traffic served by origin servers, it also reduces the traffic flowing into the client over the last-mile connection. In the case of dial-up clients, this translates into a significant reduction in end-user response times. Furthermore, for content providers who use CDN services, CSI can significantly reduce their CDN-related costs by delivering only ESI fragments, and not entire pages, from edge servers to clients.

Our performance study shows that the overhead of CSI assembly is small and is more than offset by the reduction in transmission time over the last mile. Our micro-benchmark pages, as well as experiments with two sample real pages, showed very significant net reductions in page time-to-display. Edge-side page assembly overhead, while similar to CSI, occurs *in addition* to the last mile transmission time. It therefore only adds to the total time-of-display of ESI pages, assuming that the last mile is the bottleneck.

Acknowledgements

We wish to thanks Walter Sturm for encouragement to pursue this project. We are grateful to Irina Rabinovich and Mary Fernandez for help with our Javascript and XML questions. We would also like to thank Jim Challenger and the anonymous referees for their helpful comments.

References

[1] Guide to active server pages. `http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/iisref/aspguide.htm`.

[2] Amr Awadallah and Mendel Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, 2002.

[3] C. Brabrand, A. Møller, S. Olesen, and M.I. Schwartzbach. Language-based caching of dynamically generated HTML. *World Wide Web*, 5(4):305–323, 2002.

[4] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of the 1998 Middleware Conference*, September 1998.

[5] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of INFOCOM*, pages 844–853, 2000.

[6] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, pages 83–94, December 1997.

[7] Fred Douglis, Sonia Jain, John Klensin, and Michael Rabinovich. Click-once hypertext: Now you see it, now you don't. In *Proceedings of the Second IEEE Workshop on Internet Applications*. IEEE, July 2001.

[8] Edge Side Includes W3C submission. `http://www.w3.org/Submission/2001/09/`, September 2001.

[9] Ejasent, inc. `http://www.ejasent.com`.

[10] ESI - Accelerating E-Business Applications: Overview. `http://www.esi.org/overview.html`, September 2002.

[11] JSP: Java server pages. `http://java.sun.com/products/jsp/`.

[12] Jupiter internet access model (US only). Broadband. Jupiter Media Metrix, August 2001.

[13] P. Karbhari, M. Rabinovich, Z. Xiao, and F. Douglis. ACDN: a content delivery network for applications. In *Proceedings of the ACM SIGMOD Conference, Demonstrations track*, page 619, 2002.

[14] Module mod_rewrite URL rewriting engine. `http://httpd.apache.org/docs/mod/mod_rewrite.html`.

[15] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM Conference*, pages 181–194, September 1997.

[16] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, Feb 2002.

[17] Network traffic & revenue analysis. market update. RHK, Inc., July 2002.

[18] PHP: Hypertext preprocessor. `http://www.php.net/`.

[19] W. Shi and V. Karamcheti. CONCA: An architecture for consistent nomadic content access. In *Proceedings of the Workshop on Cache, Coherence, and Consistency (WC3'01)*, June 2001.

[20] Apache tutorial: Introduction to server side includes. `http://httpd.apache.org/docs/howto/ssi.html`.

[21] Craig E. Wills and Mikhail Mikhailov. Examining the cacheability of user-requested Web resources. In *Proceedings of the 4rd Web Caching and Content Delivery Workshop*, April 1999. http://workshop99.ircache.net/Papers/wills-final.ps.gz.

[22] Xinclude. `http://www.w3.org/TR/xinclude`.

A Flexible and Efficient Application Programming Interface (API) for a Customizable Proxy Cache

Vivek S. Pai, Alan L. Cox, Vijay S. Pai, and Willy Zwaenepoel
iMimic Networking, Inc.
2990 Richmond, Suite 144
Houston, TX 77098

Abstract

This paper describes the design, implementation, and performance of a simple yet powerful Application Programming Interface (API) for providing extended services in a proxy cache. This API facilitates the development of customized content adaptation, content management, and specialized administration features. We have developed several modules that exploit this API to perform various tasks within the proxy, including a module to support the Internet Content Adaptation Protocol (ICAP) without any changes to the proxy core.

The API design parallels those of high-performance servers, enabling its implementation to have minimal overhead on a high-performance cache. At the same time, it provides the infrastructure required to process HTTP requests and responses at a high level, shielding developers from low-level HTTP and socket details and enabling modules that perform interesting tasks without significant amounts of code. We have implemented this API in the portable and high-performance iMimic DataReactor™ proxy cache[1]. We show that implementing the API imposes negligible performance overhead and that realistic content-adaptation services achieve high performance levels without substantially hindering a background benchmark load running at a high throughput level.

1 Introduction

As the Internet has evolved, Web proxy caches have taken on additional functions beyond caching Internet content to reduce latency and conserve bandwidth. For instance, proxy caches in schools and businesses often perform content filtering, preventing users from accessing content deemed objectionable. Caches in content distribution networks (CDNs) may perform detailed access logging for accounting purposes, pre-position popular items in the cache, and prevent the eviction of certain items from memory or disk storage. Support for these features has been added to caches developed in both academia and industry.

However, a proxy cache designer can not foresee all possible uses for the proxy cache and thus cannot include all features required by application implementers. While some developers of major systems such as CDNs have added their desired functionality to open-source caches, these developers are then burdened by the sheer volume of source code (over 60,000 lines in Squid-2.4 [18]). Additionally, their changes will likely conflict with later updates to the base proxy source, making it difficult to track bug fixes and upgrades effectively. Consequently, such *ad hoc* schemes erode the separation of concerns that underlies sound software engineering. The application developer should not have to reason with the details of the cache in order to add functionality. Instead, the developer should be able to write in standard languages such as C using standard libraries and system calls as appropriate for the task at hand.

One approach to enable such value-added services is to locate those functions on a separate server that communicates with the cache through a domain-specific protocol. The Internet Content Adaptation Protocol (ICAP) adopts this approach, allowing caches to establish TCP connections with servers that modify requests and responses to and from clients and origin servers [8]. On each HTTP request and response that will be modified, an ICAP-enabled proxy constructs a request which consists of ICAP-specific headers followed by the complete HTTP headers and body in chunked format. The proxy then collects a response from the ICAP server providing a complete set of modified HTTP headers and body. In addition to the TCP/IP socket overhead for communicating with the external service, such a protocol also adds overhead to parse the protocol headers and chunked data transfer format and to encapsulate HTTP messages within the protocol. Further, current implementations of ICAP locate value-added services on a separate server machine, even if the host CPU of the cache is not saturated.

An alternative approach is to use an application programming interface (API) that allows user modules to be directly loaded into the core of the cache and run ser-

[1] DataReactor is a trademark of iMimic Networking, Inc.

vices either on the cache system or on a separate server as desired. This paper presents an API that enables programming of a proxy cache after deployment. This API turns a previously monolithic proxy cache into a programmable component in a Web content delivery infrastructure, cleanly separating new functionality from the cache's core. At the same time, the API allows extremely fast communication between the cache and the user modules without the need for TCP connections or a standardized parsing scheme. Specifically, the API provides the infrastructure to process HTTP requests and responses at a high level, shielding developers from the low-level details of socket programming, HTTP interactions, and buffer management. This API can also be used to implement the ICAP standard by creating a dynamically loaded module that implements the TCP and parsing aspects of ICAP. The API extends beyond the content adaptation features of ICAP by providing interfaces for content management and specialized administration.

Several technological trends have made single-box deployment of API-enabled proxy servers more attractive. Among these are more available processing power and better OS support for high-performance servers. Proxy software in general has improved in efficiency while microprocessor speeds have increased. Recent benchmarks have shown that a 300 MHz 586-class processor is sufficient to handle over 7Mbps of traffic, enough for multiple T-1 lines [15]. Current microprocessors with two architectural generations of improvement and a clock rate that is 5-8 times higher will have significant free CPU for other tasks. Proxy servers running on general-purpose operating systems have met or exceeded the performance of appliances running customized operating systems. As a result, the proxy server has become a location that handles HTTP traffic and has the capacity and flexibility to support more than just caching.

Unlike some previously proposed schemes for extending server or proxy functionality, the API presented in this paper uses an event-aware design to conform to the implementation of high-performance proxy servers. By exposing the event-driven interaction that normally occurs within proxies, high performance implementations can avoid the overhead of using threads or processes to handle every proxy request. We believe that this performance-conscious approach to API design allows higher scalability than previous approaches, following research showing the performance advantages of event-driven approaches to server design in general [13].

We have implemented this API in the iMimic DataReactor, a portable, high-performance proxy cache. We show that implementing the API imposes negligible performance overhead and allows modules to consume free CPU cycles on the cache server. The modules themselves achieve high performance levels without substantially hindering a background benchmark load running at high throughput. While the API style is influenced by event-driven server design, the API is not tied to the architecture of any cache, and it can be deployed more widely given systems that support standard libraries and common operating system abstractions (e.g., threads, processes, file descriptors, and polling).

The rest of this paper proceeds as follows. Section 2 describes the general architecture of the system and the design of modules that access the API. Section 3 discusses the API in more detail. Section 4 describes sample modules used with the API and discusses coding issues for these modules. Section 5 provides a more detailed comparison of the API with ICAP. Section 6 describes the implementation of the API in the iMimic DataReactor proxy cache and presents its performance for some sample modules. Section 7 discusses related work, and Section 8 summarizes the conclusions of this paper.

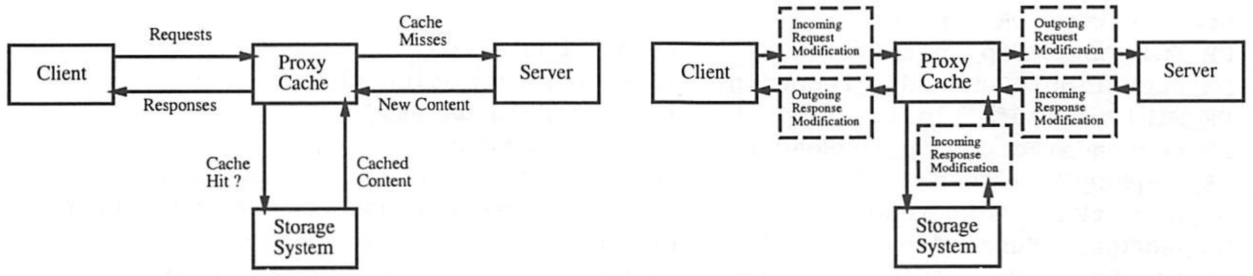
2 Structure of the API

The underlying observation that shapes the structure of the API is that a high-performance API should adopt the lessons learned from the design of high-performance server architectures. As a result, we use an event-driven approach as the most basic interaction mechanism, exposing this level directly to modules as well as using it to construct support for additional communication models based on processes or threads. By exposing the event-driven structure directly to modules, the API can achieve high scalability with minimal performance impact on the proxy. This approach dispenses with multiple thread contexts or multiple processes, enabling the scalability gains previously observed for server software [13]. The rest of this section describes how the API integrates with HTTP processing in the cache and how customization modules are designed and invoked.

2.1 Integration with HTTP Handling

Figure 1 illustrates the flow of request and response content through the proxy. Clients send requests to the proxy, either directly or via redirection through an L4/7 switch. The proxy interprets the request and compares it with the content it has stored locally. If a valid, cached copy of the request is available locally, the proxy cache obtains the content from its local storage system and returns it to the client. Otherwise, the proxy must modify the HTTP headers and contact the remote server to obtain the object or revalidate a stale copy of the object fetched earlier. If the object is cacheable, the proxy cache stores a copy of it in order to satisfy future requests for the object.

These various interactions between the proxy, the


Client
Requests
Responses
Proxy Cache
Cache Misses
New Content
Server
Cache Hit ?
Cached Content
Storage System
Incoming Request Modification
Outgoing Response Modification
Outgoing Request Modification
Incoming Response Modification
Incoming Response Modification

(a) Original proxy server (b) API-Enabled proxy server

Figure 1: Original and modified data transfer paths in a proxy server

client, and the server provide the basis for the primary functions of the API. In particular, the API provides customization modules with the ability to register *callbacks*, special functions with defined inputs and outputs that are invoked by the cache on events in the HTTP processing flow. These callback points are the natural processing points within an event-driven server; by exposing this structure, the API implementation can support modules with high scalability and low performance impact. Examples of such HTTP processing events include the completion of the request or response header, arrival of request or response body content, logging the completion of the request, and timer events.

As these interactions are common to all proxy servers, all available proxy software should be able to provide the hooks needed for implementing this API. A full implementation of this API is *http-complete*, in that it can support any behavior that can be implemented via HTTP requests and responses; this level of completeness is necessary to allow modules to perform content adaptation without limits. In addition to these hooks, the API also provides other mechanisms to control policy/behavioral aspects of the cache not covered within the scope of HTTP (e.g. prefetching content, request logging, server selection in round-robin DNS).

Modules register a set of function pointers for various events by providing a defined structure to the underlying proxy core. This structure is shown in Figure 2, and contains non-`NULL` entries for all of the callback functions of interest to the module. The module may specify `NULL` values for callback points for which it elects to receive no notification. Note that the events corresponding to the receipt of request or response body data may be triggered multiple times in an HTTP transaction (request-response pair), allowing the module to start working on the received data immediately without waiting or buffering.

The API tags each transaction with a unique identifier that is passed to each callback as the first argument. This allows the module to invoke multiple interactions with the proxy for a single HTTP transaction while recognizing which parts of the transaction have already taken place. Additionally, the API allows the module to pass back a special "opaque" value after completing each callback. This opaque value is not interpreted by the cache itself, but is instead passed directly to the next API callback function for this transaction. The opaque value typically contains a pointer to a data structure in the module specific to this transaction. The API invokes the `dfp_opaquefree` callback function when the transaction is complete, allowing the module to clean up the underlying structure as desired. The client IP address is also an input to each HTTP-related callback, allowing for different decisions depending on the source.

If a module is no longer interested in event notifications for a particular transaction, it may return a special response code recognized by the API. This allows modules to cease further effort on a transaction if, for example, request or response headers indicate that the module's service is not applicable. Once a module returns that response code, the API will not invoke any more callbacks for that transaction, but will invoke callbacks for other transactions.

2.2 Module Design

API modules are precompiled object files that are either dynamically linked into the proxy or are spawned in a separate address space. For security reasons, clients of the proxy cannot install modules into the proxy. Modules are trusted software components that must be installed by an administrator with the authority to configure the cache.

Modules export a set of standard entry points that are used by the proxy cache to invoke methods in the module in response to certain events affiliated with HTTP processing. The internal design of modules is not restricted; they can spawn other programs, invoke interpreted code, or call standard library and operating system APIs without impediment. The latter are particularly useful, for ex-

```
typedef struct DR_FuncPtrs {
  DR_InitFunc *dfp_init;                  // on module load
  DR_ReconfigureFunc *dfp_reconfig;       // on configuration change
  DR_FiniFunc *dfp_fini;                  // on module unload
  DR_ReqHeaderFunc *dfp_reqHeader;        // when request header is complete
  DR_ReqBodyFunc *dfp_reqBody;            // on every piece of request body
  DR_ReqOutFunc *dfp_reqOut;              // before request goes to remote server
  DR_DNSResolvFunc *dfp_dnsResolv;        // when DNS resolution needed
  DR_RespHeaderFunc *dfp_respHeader;      // when response header is complete
  DR_RespBodyFunc *dfp_respBody;          // on each piece of response body
  DR_RespReturnFunc *dfp_respReturn;      // when response returned to client
  DR_TransferLogFunc *dfp_logging;        // logging entry after request done
  DR_OpaqueFreeFunc *dfp_opaqueFree;      // when each response completes
  DR_TimerFunc *dfp_timer;                // periodically called for maintenance
  int dfp_timerFreq;                      // timer frequency in seconds
} DR_FuncPtrs;
```

Figure 2: Structure provided by modules to register callback functions for specified events.

ample, for communicating with other systems via TCP/IP connections to provide services desired by the module.

Multiple modules can be active, and modules can be dynamically loaded and unloaded. Cascading multiple modules allows developers to combine services such as content filtering with image transcoding for a wireless business environment or site monitoring and content-preloading for a content delivery network. The ability to dynamically load and unload modules allows policies such as deactivating content filtering outside of normal work hours while still using image transcoding. The module programmer or deployer must specify the order of invocation for multiple modules so that data arrives as expected and interactions remain sensible.

2.3 Execution Models

The API supports modules executing in several formats, including processes, threads, and callbacks. The module sees the same interfaces in all cases, but the underlying implementations may differ. Since all of the models present the same interface, module developers are free to change the model used as the performance or flexibility needs of their modules change.

Processes – The most flexible model for an API module is to use a (Unix) process. In this manner, all module processing takes place in a separate address space from the proxy core, and the module is at liberty to use any operating system interfaces, run other programs, communicate across the network, perform disk operations, or undertake other slow or resource-intensive operations. A process may be single-threaded or multi-threaded. Using the process model enables the trivial use of multiprocessors and, with appropriately written modules, the ability to harness clusters of machines in a network. The flexibility of the process model implies more overhead in the operating system, including extra memory for storing the process state and more CPU overhead when the OS-supplied interprocess communication mechanisms are invoked to exchange information between the proxy core and the module. However, these communication costs are relatively minor for modules that perform significant processing.

Threads – Threads provide a higher-performance alternative to processes. In this model, the module spawns multiple threads in the proxy's address space. Each thread requires less overhead than a full process and the use of shared memory allows higher performance communication between the module and the proxy cache. Like the process model, threads can also trivially take advantage of multiprocessors. However, since the threads share the address space with the proxy cache, they must be careful not to corrupt memory or invoke system calls that affect the state of the proxy itself.

Callbacks – Callbacks are the lowest overhead mechanism for content adaptation, since the module is directly linked into the proxy cache's address space and invoked by the proxy cache state machine. As a result, callback overhead is comparable to a single function call. Since callbacks are performed synchronously in the proxy, the module's routines should not perform any blocking operation such as opening files, waiting on network operations, or synchronously loading data from disk. Nevertheless, callbacks may invoke nonblocking network socket operations and use polling functions provided by the API to determine when data is available for them. Modules that fit these criteria may use callbacks for the highest perfor-

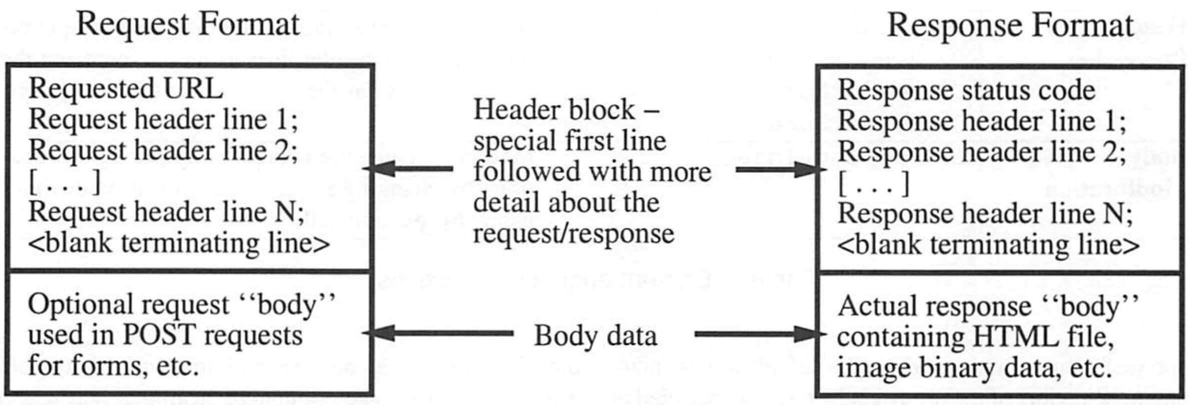


Figure 3: Structure of HTTP requests and responses

mance. Modules using callbacks should also be careful to avoid corrupting memory or performing stray pointer accesses, since corrupting memory can affect the running of the proxy cache.

3 API Functions

This section describes the functions provided by the API. This section does not aim to be a full manual, but rather describes the reasons for specific functions and the types of tasks supported by the API.

3.1 Content Adaptation Functions

The content adaptation functions allow modules to inspect and modify requests and replies as they pass through the proxy cache. Since any portion of the transfer can be inspected and modified by the modules, the API provides a powerful mechanism to use the proxy cache in a variety of applications. For example, the proxy cache can be used as a component of a Content Distribution Network (CDN) by developing a module that inspects the user's requested URL and rewrites it or redirects it based on geographic information. In mobile environments where end users may have lower-resolution displays, an API module could reduce image resolution to save transmission bandwidth and computation/display time on low-power devices. Such modular solutions would consist of far less code and better defined interactions than modifications to open-source code.

As explained in Section 2, the mechanisms for the API are closely integrated with the processing of requests and responses in the HTTP protocol. In the HTTP protocol, requests and responses have a well-defined structure, shown in Figure 3. Each consists of a header block with a special request/response line followed by a variable number of header lines, and then a variable-length data block.

The content adaptation interfaces allow modules to specify entry points that are called when the proxy handles the various portions of the requests and responses. Some of the entry points are called only when a particular portion is complete (e.g., when all of the response headers have been received), while others are called multiple times (e.g., as each piece of body data is received on a cache miss). For example, a module could provide routines that examine and modify the first line of each request and the full contents of each response that pass through the proxy.

The API includes header processing routines that allow searching for particular HTTP headers, adding new headers, deleting existing headers, and declaring that the current set of headers is completed and may be sent on. These routines provide a simple and flexible interface for the module to customize the HTTP-level behavior of the request, in isolation or in conjunction with transformation of the content body. This approach also insulates the modules from the details of the HTTP protocol, shifting the burden of providing infrastructure onto the proxy. Table 1 lists the routines provided for header and body manipulation.

The cache may also store modified content, allowing it to be served without requiring adaptation processing on future requests. This can be accomplished by registering interest in the arrival of the response header, and modifying the cache control header appropriately, causing the proxy cache to store the modified content if it would not be stored by default. When content adaptation is performed based on features of the client's request, the module can use the HTTP `Vary` header to indicate multiple variant responses cached for the same URL.

3.2 Content Management Functions

The API's content management features allow modules to perform finer-grained control over cache content than a proxy cache normally provides. These routines are ap-

Header Processing	`DR_HeaderDelete`
	`DR_HeaderFind`
	`DR_HeaderAdd`
	`DR_HeaderDone`
		Modules can examine and modify the multiple request/response header lines as they arrive from the client/server, or as they are sent out from the proxy.

Body Modification	`DR_RespBodyInject`	Modules change the content body during notifications by changing return values, or at other times using this explicit call.

Table 1: Content adaptation functions

propriate for environments where the default content management behaviors of the proxy cache must be modified or augmented with information from other sources. The routines in this area fall into three broad categories: content freshness modification and eviction, content preloading, and content querying. The related functions are shown in Table 2.

Server accelerators and surrogates in content distribution networks benefit from finer-grained control over content validation to improve response time, reduce server load, or provide improved information freshness. In particular, surrogates normally achieve the maximum benefit when the origin servers set large expiration times to reduce the frequency of revalidation. However, the proxy may not see the most recent content if the underlying content changes during this time. When an external event (such as a breaking news item) occurs that voids the expiration information, the content management routines can update the proxies by invalidating cached content and fetching the newest information. The content management routines thus allow programmed, automatic control for exceptional situations while still using the regular mechanisms of the proxy cache under normal circumstances.

The content management routines also allow programmatic and dynamic preloading of objects into the cache, in addition to the content prepositioning support that already exists in many proxy caches. Preloading modules may be coupled to information sources outside of the proxy rather than just based on timer information. For example, a service provider running a network of caches could create a custom module to inject documents into caches as their popularity increases in other parts of the network.

Finally, the content management controls can also be used to provide "premium" services to portions of cached content. For example, the content management controls can be used to periodically query the cache and refresh the pages of premium customers, changing their cache eviction behavior.

Even though these functions change the material stored in the cache, the API introduces no security issues because its functions are invoked directly from trusted modules running on the same machine. However, the modules themselves may need to implement a security policy, especially if they take commands from external sources. The modules may use standard library functions to provide the mechanisms for this security; the API need not provide mechanisms since none of the encryption or authentication required is specific to the proxy cache environment.

3.3 Customized Administration

The API provides notifications that can be used to augment the administrative and monitoring interfaces typically available in proxy caches through Web-based tools, command-line monitoring, and SNMP. The event notification for logging allows the proxy to extract real-time information and manipulate it in a convenient manner in order to feed external consumers of this information. For example, content distribution networks and web hosting facilities may use real-time log monitoring to provide up-to-date information to customers about traffic patterns and popularity on hosted web sites, or to notice high rates of "page not found" errors and dynamically create pages to redirect users to the appropriate page. While all of this information can be obtained in off-line post-processing of access logs, the API allows deployers to implement custom modules that meet their specific needs for realtime analysis and action.

The administrative interfaces of the API can also be used to monitor networks of proxies, and to combine this information with other sources in Network Operation Centers (NOCs). For example, if a company is running a network of proxies and notices an unusually low request rate at one such proxy, this information can be combined with other information to help diagnose congested links or overloaded systems beyond the company's control. Since the proxy is functioning normally, one isolated measurement provides less reliable information than programmatic and comparative measurement of different systems.

3.4 Utility Functions

Table 3 describes utility functions provided by the API. The `DR_FDPoll` family of functions allow callback-

Naming & Identification	DR_ObjIDMake DR_ObjIDRef DR_ObjIDUnref	Modules use internal identifiers for content management functions rather than requiring full URLs in all calls. These functions are used to create/destroy the identifiers
Existence & Lifetime Handling	DR_ObjQuery DR_ObjFetch DR_ObjValidate DR_ObjExpirationSet DR_ObjDelete	For querying the existence/property of objects and loading, refreshing, and deleting them as needed
Cache Injection	DR_ObjInject DR_ObjInjectStart DR_ObjInjectBody DR_ObjInjectDone	Objects can be manually added into the proxy without having the proxy initiate a fetch from a remote server. The object can be injected all at once, or in pieces.
Data Reading	DR_ObjRead DR_ObjReadStart DR_ObjReadPart DR_ObjReadDone	Objects in the cache can be loaded into memory allocated by the module without being requested by the client. The object can be read all at once or in pieces.

Table 2: Content management functions

External Communication	DR_FDPollReadNotify DR_FDPollReadClear DR_FDPollWriteNotify DR_FDPollWriteClear DR_FDPollClose	Modules may communicate externally using sockets, even when using the callback mechanism. In this case, they use asynchronous operations and use these functions to have the main proxy notify them of events on their sockets.
Custom Logging	DR_StringFromStatus DR_StringFromLogCode DR_StringFromLogTimeout DR_StringFromLogHier	These functions provide the standard text strings used for the extended log format.
Configuration	DR_ConfigOptionFind	User modules invoke this function to extract information from the cache configuration file.

Table 3: Utility functions

based modules to use nonblocking socket operations. These functions allow external communication without blocking and without requiring modules to implement their own event management infrastructure; instead, such modules can simply utilize the underlying event mechanisms of the OS and host proxy. In particular, the `ReadNotify` and `WriteNotify` calls register functions that should be invoked when an event (either data available to read or space available to be written, respectively) takes place on a given file descriptor. The `ReadClear` and `WriteClear` functions indicate that the previously registered function should no longer be called, and the `Close` function indicates that the file descriptor in question has been closed and all currently registered notifications should be deregistered. All of these functions can be implemented using `poll`, `select`, `kevent`, `/dev/poll`, or other OS-specific mechanisms; their only requirement is that the underlying OS support the notion of file descriptors and some form of event notification, which are common to all standard systems.

The custom logging functions provide canonical names for the codes passed by the proxy to module functions for log notifications. `DR_ConfigOptionFind` allows the user module to extract desired information from the cache configuration files.

4 Sample Modules

To experiment with the API, both from a functional aspect as well as from a performance perspective, we developed some modules that use various aspects of the interface, including a module that implements ICAP. We were pleased with the simplicity of module development and the compactness of the code necessary to implement various features. Initial development and testing of each module required from a few hours to a few days. More detail about each module's behavior and implementation is pro-

Module Name	Total Lines	Code Lines	Semicolons	# API call sites
Ad Remover	175	115	51	4
Dynamic Compressor	387	280	126	11
Image Transcoder (+ helper)	391 + 166	309 + 118	148 + 54	10
Text Injector (+ helper)	473 + 56	367 + 32	170 + 8	12
Content Manager	675	556	289	56
ICAP client	1024	719	321	15

Table 4: Code required to implement sample services

vided below. Table 4 summarizes information about the code size needed for each module. Since the modules are freed from the task of implementing basic HTTP mechanisms, none of them are particularly large. The "Total Lines" count includes headers and comments, the "Code Lines" count removes all blank lines and comments, and the "Semicolons" count gives a better feeling for the number of actual C statements involved. All modules use the callback interface, with some spawning separate helper processes under their control.

Ad Remover – Ad images are modified by dynamically rewriting their URLs and leaving the original HTML unmodified. On each client request, the module uses a callback to compare the URL to a known list of ad server URL patterns. Matching URLs are rewritten to point to a cacheable blank image, leading to cache hits in the proxy for all replaced ads. To account for both explicitly-addressed and transparent proxies, the module constructs the full URL from the first line of the request and the `Host` header line of the request header. On modified requests, the `Host` header must be rewritten as well, utilizing the `DR_Header*` functions. Other uses for this module could include replacing original ads with preferred ads.

Dynamic Compressor – This module invokes the `zlib` library from callbacks to compress data from origin servers and then caches the compressed version. Clients use less bandwidth and the proxy avoids compressing every request by serving the modified content on future cache hits. This module checks the request method and `Accept-Encoding` header to ensure that only GET requests from browsers that accept compressed content are considered. The response header is used to ensure that only full responses (status code 200) of potentially compressible types (non-images) are compressed. The header is also checked to ensure that the response is not already being served in compressed form and is not an object with multiple variants (since one of those variants may already be in compressed form). Using the `DR_Header*` functions, the outbound response must be modified to remove the original `Content-length` header and to insert a `Vary` header to indicate that multiple versions of the object may now exist.

Image Transcoder – All JPEG and GIF images are converted to grayscale using the `netpbm` and `ijpeg` packages. Since this task may be time-consuming, it is performed in a separate helper process. The module buffers the image until it is fully received, at which point it sends the data to the helper for transcoding. The helper returns the transcoded image, or the original data if transcoding fails. The module kills and restarts the helper if the transcoding library fails, and also limits the number of images waiting for transcoding if the helper can not satisfy the incoming rate of images. The module uses the `DR_FDPoll*` functions to communicate with the helpers, the `DR_Header*` functions to modify the response, and the `DR_RespBodyInject` function to inject content into an active connection.

Text Injector – The main module scans the response to find the end of the HTML `head` tag, and then calls out to a helper process to determine what text should be inserted into the page. The helper process currently only responds with a text line containing the client IP address, but since it operates asynchronously, it could conceivably produce targeted information that takes longer to generate. The module passes data back to the client as it scans the HTML, so very little delay is introduced. For reasons similar to the case of the Image Transcoder module, the `DR_FDPoll*`, `DR_Header*`, and `DR_RespBodyInject` functions are all invoked.

Content Manager – This demonstration module accepts local `telnet` connections on the machine and presents an interface to the `DR_Obj*` content management functions. The administrator can query URLs, force remote fetches, revalidate objects, and delete objects. Object contents can also be displayed, and dummy object data can be forced into the cache. The module uses the `DR_FDPoll*` family of functions to perform all processing in callback style even while waiting on data from network connections.

ICAP Client – This module implements the ICAP 1.0 draft for interaction with external servers that provide value-added services [8]. The module must encapsulate HTTP requests and responses in ICAP requests, send those requests to ICAP servers, retrieve and parse responses, and send forth either the original HTTP message or the modified message provided by the ICAP server. All of this processing can be implemented through callbacks with the assistance of the polling functions for network event notification. Implementing ICAP as a module rather than an integrated part of the proxy core is particularly appropriate as ICAP specifications continue to evolve.

5 Comparison with ICAP

This section discusses key points of comparison between our API and the Internet Content Adaptation Protocol (ICAP) draft specification [8]. We do not seek to undertake a detailed quantitative performance comparison since the ICAP standard and implementations are still evolving. We instead focus on differences in functionality and mechanism.

Functionality. ICAP provides services for content-adaptation, while the API additionally provides services for content-management and customized administration. As a result, ICAP provides a useful component for content delivery, but cannot enable the more detailed management infrastructure for a content delivery network without changes to the underlying cache architecture. In contrast, the API provides services to push, query, invalidate, or modify data while it is in the cache, and also provides features for real-time monitoring and integration with statistics. Some of these features may even inform content-adaptation; for example, real-time monitoring can potentially provide additional information beyond cookies alone (e.g., frequency of connections from a client IP address or authenticated user) that may lead to different transcoding behavior or object freshness policy.

Additionally, the polling functions of Section 3.4 encapsulate the low-level details of concurrency management. These polling functions enable the API modules to efficiently use the underlying OS and cache features for socket I/O to a variety of network services (including external ICAP servers) while avoiding the programming difficulty of implementing such an event notification state machine directly.

Mechanism. The primary differences in content adaptation mechanism between ICAP and the API stem from the communication methods used. ICAP invokes all communication by having the cache initiate contact with the service through a TCP/IP socket. In contrast, the API allows the cache to directly invoke functions registered to provide a service. While current ICAP implementations locate the value-added services on a separate server, the API allows for the use of either a separate server or a cache-integrated module. The latter is particularly valuable as processor speeds continue to accelerate faster than all other parts of the system, enabling substantial additional services beyond caching without saturating the CPU. The API is also sufficiently flexible to implement ICAP as a module rather than part of the proxy core.

ICAP allows servers to statically inform proxies that HTTP data for certain file extensions should not be passed to them, that others should be sent for *previewing*, and that others should be always sent in their entirety. For those HTTP requests and responses that must be previewed, an ICAP-enabled proxy constructs a preview message consisting of ICAP-specific headers followed by the complete HTTP headers and some arbitrary amount of the HTTP body. The ICAP server then indicates whether or not the proxy should continue sending body data for modification. If so, or if the file extension indicates that this request should always be sent rather than previewed, the proxy must send the entire set of HTTP headers and body. The server will then respond either with an indication that no modifications will take place or with a complete set of modified HTTP headers and body. The primary goal of previewing is to allow the service to act upon a message by reading the HTTP headers, but ICAP requires the proxy to construct ICAP headers and encapsulate the HTTP headers on a preview, after which it must parse a response from the ICAP server. In contrast, the API allows for more direct header examination with `DR_HeaderFind`, requiring no higher-level ICAP wrapper headers.

Services should also have an easy mechanism to decide that they have no further interest in an HTTP message. ICAP provides no mechanism to continue past the preview and then stop adaptation before seeing the full body. The API allows the service to dynamically turn off interest in further callbacks for a transaction at any point in the headers or body. This difference could affect the text injector module of Section 4, since the text injection process might finish at any arbitrary point in the body.

In short, while ICAP can provide a variety of useful content-adaptation features, the API presented here exposes an interface that provides a superset of these functions while also enabling low-overhead coordination with service modules (including ICAP itself).

6 Implementation and Performance

In this section, we describe the implementation of the API in the iMimic DataReactor proxy cache and conduct a series of experiments to understand various performance scenarios related to the API. We measure the impact of adding API support into the DataReactor, the various costs of API features, and the performance of some of the sample modules that use the API.

6.1 The iMimic DataReactor Proxy Cache

Evaluating the impact of the API is heavily dependent on the quality of the underlying platform. A slow proxy will mask the overhead of the API, while a fast one will more easily expose the additional latency resulting from the API. The iMimic DataReactor Proxy Cache is a commercial high-performance proxy cache. It supports the standard caching modes (forward, reverse, transparent) and is portable, with versions for the x86, Alpha, and Sparc architectures and the FreeBSD, Linux, and Solaris operating systems. It has performed well in the vendor-neutral Proxy Cache-Offs, setting records in performance, latency, and price/performance [15, 16, 17].

We test forward-proxy (client-side) cache performance using the Web Polygraph benchmark program running a modified version of the Polymix-3 workload [16]. We use this test and workload because it has the highest number of independently-measured entries of any web proxy benchmark, and it heavily stresses proxy server performance. For the sake of time, we shorten our performance tests to use a 2 hour load plateau instead of four hours, and fill the disks only once before all tests rather than before each test. These changes shorten the load phase of the Polygraph test to roughly 6 hours instead of 10.5, and avoiding a separate fill phase reduces the length of each test by an additional 10-14 hours. The primary performance impact is a 3–4% higher hit ratio than an official test because of a smaller working set and data set. We call this modified test PMix-3.

Polygraph stresses various aspects of proxy performance, particularly in network and disk-related areas. It uses per-connection and per-packet delays between the proxy and simulated remote servers to cause cache misses to have a high response time. Likewise, it generates data sets and working sets that far exceed physical memory, causing heavy disk access. Polygraph stresses connection management by scaling the number of simulated clients and servers with the request rate.

The test system runs FreeBSD 4.4 and includes a 1400 MHz Athlon, 2 GB of memory, a Netgear GA-620 Gigabit Ethernet NIC, and five 36 GByte 10000 RPM SCSI disks. All tests use a target request rate of 1450 $\frac{\text{requests}}{\text{second}}$. This throughput compares favorably with other high-end commercial proxy servers and is over a factor of ten higher than what free software has demonstrated [16]. At this rate, the proxy is managing over 16000 simultaneous connections and 3600 client IP addresses. Given the fixed request rate, this test demonstrates any latency differences in the various test scenarios. (Polygraph also shows some run-to-run randomness in the offered workload, leading to additional minor variations.)

6.2 API Implementation Overhead

To understand the performance overhead of implementing the API in the DataReactor, we start with a standard DataReactor platform, incrementally add features, and test the result. Overheads from implementing the API result in increased hit and miss response times, since throughput is kept constant. Table 5 lists the results for these tests.

The various columns of Table 5 are as follows: "Baseline" is the standard DataReactor software without API support. "API-Enabled" is the same software with API support, but without any modules loaded. "Empty Callback" adds a module with all notifications specified, but with no work done in any of them. "Add Headers" adds extra headers to all inbound/outbound paths on the proxy, so four extra headers will be introduced on each transaction. "Body + Headers" additionally copies the response body of each reply and overwrites the response body with this copy.

The "API-Enabled" test shows that implementing the API adds virtually no overhead on cache hits and only a small overhead on cache misses. Actually installing a module causes a slight slowdown on hits and misses due to the extra calls needed. Due to the extremely small hit times, this effect appears as a 5% increase on hit time. On cache misses, where most of the time is spent waiting on the remote server, the overhead is less than one-tenth of one percent. These low overheads confirm the premise that an explicitly event-aware API design can enable an extensible proxy with minimal performance impact.

We also observe that using the features of the API, such as adding headers or modifying the body, generates low overhead. Adding headers introduces some extra delay on misses, but even modifying the full body does not generate any significant spike in response times. The hit times for "Body + Headers" show a 6% increase over the "Empty Callback", which translates into a cumulative 11.5% increase versus the baseline. However, in absolute terms, the increase is less than 2.5ms, or less than 1% of the overall response time.

6.3 Performance Methodology

We construct several tests to assess the performance of some of our content-adaptation modules, both on their own and in terms of their impact on the overall system. However, we cannot rely solely on Pmix-3 to generate the load, since this workload does not generate realistic content for the objects in its test. Without realistic content, measuring the performance of some of our content adaptation modules would be meaningless. For example, the Image Transcoder module would fail to perform any transcoding, and would return the images unmodified.

	Baseline	API Enabled	Empty Callback	Add Headers	Body + Headers
Throughput (reqs/sec)	1452.87	1452.75	1452.89	1452.62	1452.84
Response time (ms)	1248.99	1248.95	1251.25	1251.98	1250.14
Miss time (ms)	2742.53	2743.18	2744.33	2745.07	2746.98
Hit time (ms)	19.82	19.86	20.87	20.85	22.10
Hit ratio (%)	57.81	57.81	57.76	57.74	57.85

Table 5: Performance tests to determine overhead of implementing API

	Baseline	Ad Remover	Images 25 Trans/s	Images Max Trans	Max Trans nice 19	Compress 75 obj/s	Compress 95 obj/s
Throughput (reqs/s)	1452.87	1452.72	1452.65	1452.73	1452.68	1452.73	1452.88
Response time (ms)	1248.99	1248.87	1256.60	1277.76	1250.69	1252.24	1258.34
Miss time (ms)	2742.53	2743.55	2753.47	2778.09	2744.60	2745.63	2752.63
Hit time (ms)	19.82	20.42	23.21	43.30	20.15	23.44	28.69
Hit ratio (%)	57.81	57.81	57.74	57.80	57.78	57.81	57.78

Table 6: Background Pmix-3 benchmark performance when run simultaneously with content adaptation modules

Since transcoding is a more CPU-intensive process than rejecting non-image objects, the real performance impact of the transcoder could not be measured.

For the image transcoding and dynamic compression tests, we extend the Polygraph simulation testbed with a non-Polygraph client and server to generate requests and serve real objects as responses. The new server also generates only non-cacheable responses so that the modules must be invoked on each response. The content adaptation modules identify responses from the "real" server and only consider those responses as candidates for transcoding. While this approach generates some extra load on the module versus screening out all Polygraph client requests early, we feel our approach will yield more conservative performance numbers. We also continue running a Pmix-3 test against the same proxy at the same time, and keep the Pmix-3 request rate the same as in the earlier tests for an accurate comparison.

6.4 Module Performance

Table 6 shows the performance effects of the various content adaptation modules. The "Baseline" column shows our baseline performance with no API support. The "Ad Remover" column shows the performance of the Ad Remover module examining Polygraph traffic. The next three columns show proxy performance when the image transcoder is running in different scenarios. The final columns show the Dynamic Compressor serving a certain rate of compressed objects.

The Ad Remover tests show virtually no degradation in performance. This result is not surprising, because most of this module's work consists of inspecting request headers, which is computationally cheap. This module only rewrites headers on matching URLs, and this workload does not have any URL matches.

The Image Transcoder tests show how this module can affect the overall performance of the proxy, but also how a simple change can eliminate almost all of its negative impact. Since all transcoding is performed in a helper process, we show several scenarios for this module to gain a better understanding of how it behaves. On an idle machine, the transcoder can process JPEGs of size 8 KBytes at a rate of roughly 110 per second. During the load plateau of Pmix-3, most of the CPU is utilized serving regular traffic, and less time is available to the transcoder. At this point, if we run the transcode client in infinite-demand mode, we achieve an average of 30 transcodes/sec, with a range of 20-38. When this occurs, the proxy CPU has no idle time. Transcoding at 25 reqs/sec shows an 11ms increase in miss time and a 3ms increase in hit time. When the client runs in infinite-demand mode, miss times increase by 36ms while hit times rise by 23ms.

The transcoder's negative side effects on Pmix-3 traffic suggest that the proxy and helper are competing for the CPU. This competition can be almost completely eliminated by changing the process scheduling priority (the "nice" value) of the helper to 19, giving it the lowest priority of the system. With this change, the helper runs only when the CPU is idle. For the infinite-demand workload, queues between the proxy and the helper process never become overly long since further requests are delayed until earlier responses complete. As a result, the transcoder processes all requests made to it and the system is work-conserving. Since the system is work-conserving and the CPU has idle time available, the priority change for the helper process only affects the scheduling of the helper but does not otherwise affect its throughput. With this simple change, the Pmix-3 performance numbers return to values only slightly worse than the base proxy.

On an idle machine, the dynamic compressor module can satisfy approximately 400 compressions per second with the input data as an 8 KByte text file of C source code. When run in combination with Pmix-3, the dynamic compressor is shown with two different workloads: compressing 75 objects per second and 95 objects per second. The system supports the lighter compression workload with very little impact on the hit or miss response time of the background Pmix-3 traffic. The heavier compression workload leads to about 10 ms increase in both miss and hit time relative to the baseline performance; however, even this still leads to less than 1% degradation of mean response time. No substantially higher rate is possible because the CPU is saturated when the Pmix-3 load plateau occurs simultaneously with 95 compressions per second.

These performance results show that the API can enable content-adaptation services to consume spare CPU cycles on the proxy cache without interfering substantially with the performance observed by transactions for unmodified content.

7 Related Work

Sections 1 and 5 discuss the Internet Content Adaptation Protocol (ICAP) and compare it with the API presented here. This section discusses other related work.

The event-aware nature of our API is clearly motivated by previous research on event-driven servers [3, 19], particularly by work showing their scalability benefits versus traditional multi-threaded or multi-process servers [13]. Through the use of dynamic loading of modules coupled with an event-driven proxy core, our implementation achieves performance comparable to adding existing states into a event-driven server.

The TransSend/TACC proxy [6] performs content adaptation using a system akin to Unix pipes, where thread-based modules receive a stream of bytes from the main proxy. In comparing the relevant section of that work, we find differences in architecture and coverage. By exposing an event-aware API, modules can choose to avoid the overhead of threads or processes, yielding higher scalability. In terms of coverage, since our API is specifically designed for caching proxy servers, it contains content management and utility functions not present in other APIs.

Commercial proxy caches by Inktomi and Novell have previously announced APIs. No public documentation of functionality or performance is available for the Inktomi API. The Novell Filter Framework provides a content adaptation system for Novell Border Manager and Volera Excelerator [12]. Filter modules are supported using only a callback model. Additionally, the system appears to be tightly integrated with the operating system kernel because standard libraries for memory management such as `malloc` are not available; instead, all memory allocation and management must take place through kernel-style memory chains. Filter Framework was never fully implemented, and has now been discontinued.

Many academic studies and commercial products have been based on modifying the source code of the Harvest cache and its successors such as Squid [3, 18]. However, if these source code changes are not integrated into the public releases of the proxy, the groups maintaining the modified proxy must track the public releases to incorporate bug fixes, performance improvements, and new features. In contrast, changes to an API-enabled proxy server only affect modules if the API specification changes.

Research in content adaptation has often shown the difficulty in modifying proxy behavior. For example, Chi et al. describe a proxy server that modifies Squid to compress incoming data objects, but keeps the original content-length header intact [4]. That work tests the proxy with a modified client that ignores the content-length header. In an API-based solution, deleting or changing headers is a simple task since the API provides the needed infrastructure.

The ad insertion proxy developed by Gupta and Baehr uses special header lines that provide information about what parts of an HTML document are ads that can be replaced by the proxy in cooperation with the origin server [7]. Their non-caching proxy was developed specifically for this purpose. The same system could be developed with an API-enabled proxy with much less effort, as the ad replacement module could use the same special headers to communicate with cooperating servers without modifying the infrastructure for managing other HTTP headers.

Various researchers have examined the issue of content management, often to address the limitations of the HTTP protocol's handling of object expiration/staleness or to take advantage of regional proxies. The PoliSquid server develops a domain-specific language to allow customization of object expiration behavior [1]. The Adaptive Web Caching project uses proxies in overlapped multicast groups to push content and perform other optimizations related to object placement [5]. Likewise, the approach proposed by Rabinovich et al. uses routing/distance information to determine when proxies should contact neighbors versus when they should request objects directly [14]. In all of these cases, a content-management

API would reduce development work of the customizers and would allow them to focus on their policies and improvements rather than the underlying mechanism.

Researchers and companies have also examined mechanisms for extending proxy server functionality using Java. The Active Cache project associates with each cacheable document a Java applet that is invoked when a proxy accesses the document [2]. Likewise, the JProxyma proxy uses Java plug-ins for performing content adaptation [9]. We believe that the API we propose can enable either approach; in particular, the use of helper processes in sample modules such as the transcoder shows that extended services can effectively launch external programs for their API interactions.

Component-based software architectures are rapidly gaining popularity in various domains of the computer industry. For instance, the applications in office productivity suites, such as Microsoft Office or the public-domain Koffice, all follow the component-based paradigm, exporting a set of APIs to other applications [10, 11]. The reason for this growing popularity is identical to the one that caused us to develop an API for proxy caches: providing the ability to control an application without having to modify it.

8 Conclusions

The need for Web proxy caches to provide a wide-ranging and growing set of services motivates systems that allow development of customization modules while shielding developers from the details of the underlying cache. This paper describes an Application Programming Interface (API) for proxy caches that allows them to become programmable components in sophisticated content delivery infrastructures. Using this API, functionality can be added to the proxy cache after it is deployed, without needing third-party source code modifications that may be difficult to maintain or needing external servers to run these services even if the cache CPU is not fully utilized. The API supports content adaptation, content management, and customized administration, and can be used to implement the ICAP Internet draft. We have demonstrated the power and ease of use of the API with some examples that show interesting tasks being performed with small amounts of code.

Although the API is independent of any particular proxy cache core, we have discussed the implementation and performance of the API in the iMimic DataReactor proxy cache. Because the API design follows the event-driven structure of a typical proxy, implementing the API has no significant effect on the performance of the proxy cache. Further, our results show two realistic content-adaptation services achieving high performance levels without substantially hindering a background benchmark load run at a high throughput level. This allows such services to consume free CPU cycles on the cache system itself, an increasingly available commodity as processor speeds accelerate faster than other portions of the system and as small-scale multiprocessors become more common.

References

[1] J. Barnes and R. Pandey. Providing Dynamic and Customizable Caching Policies. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS 99)*, Oct. 1999.

[2] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.

[3] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[4] C. Chi, J. Deng, and Y. Lim. Compression proxy server: Design and implementation. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS 99)*, Oct. 1999.

[5] S. Floyd, V. Jacobson, and L. Zhang. Adaptive web caching. In *Proceedings of the Second International Web Caching Workshop (WCW '97)*, 1997.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[7] A. Gupta and G. Baehr. Ad insertion at proxies to improve cache hit rates. In *Proceedings of the 4th International Web Caching Workshop*, Apr. 1999.

[8] ICAP Protocol Group. ICAP: the Internet Content Adaptation Protocol. Internet draft, June 2001.

[9] Intellectronix LLC. Jproxyma. http://www.intellectronix.com/jpro/aboutjpro.htm.

[10] KOffice Project. Koffice. http://www.koffice.org/.

[11] Microsoft Corporation. Microsoft Office. http://www.microsoft.com/office/.

[12] Novell Developer Kit. *Filter Framework*, Jan. 2001.

[13] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, pages 199–212, June 1999.

[14] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proceedings of the Third International Web Caching Workshop (WCW '98)*, June 1998.

[15] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. Raw data and independent analysis at http://www.measurement-factory.com/results/, Dec. 2001.

[16] A. Rousskov and D. Wessels. The third cache-off. Raw data and independent analysis at http://www.measurement-factory.com/results/, Oct. 2000.

[17] A. Rousskov, D. Wessels, and G. Chisholm. The second IRCache web cache-off. Raw data and independent analysis at http://cacheoff.ircache.net/, Feb. 2000.

[18] D. Wessels et al. The Squid Web Proxy Cache. http://www.squid-cache.org.

[19] Zeus Technology Limited. Zeus Web Server. http://www.zeus.co.uk.

NPS: A Non-interfering Deployable Web Prefetching System

Ravi Kokku Praveen Yalagandula Arun Venkataramani Mike Dahlin
Department of Computer Sciences, University of Texas at Austin
{rkoku, ypraveen, arun, dahlin}@cs.utexas.edu

Abstract

We present NPS, *a novel non-intrusive web prefetching system that (1) utilizes only spare resources to avoid interference between prefetch and demand requests at the server as well as in the network , and (2) is deployable without any modifications to servers, browsers, network or the HTTP protocol. NPS's self-tuning architecture eliminates the need for traditional "thresholds" or magic numbers typically used to limit interference caused by prefetching, thereby allowing applications to improve benefits and reduce the risk of aggressive prefetching.*

NPS avoids interference with demand requests by monitoring the responsiveness of the server and accordingly throttling the prefetch aggressiveness, and by using TCP-Nice, a congestion control protocol suitable for low priority transfers. NPS avoids the need to modify existing infrastructure by modifying HTML pages to include Javascript^{TM} code that issues prefetch requests and by wrapping the server infrastructure with several simple external modules that require no knowledge of or no modifications to the internals of existing servers. Our measurements of the prototype under a web trace indicate that NPS is both non-interfering and efficient under different network load and server load conditions. For example, in our experiments with a loaded server with little spare capacity, we observe that a threshold-based prefetching scheme causes response times to increase by a factor of 2 due to interference, whereas prefetching using NPS decreases response times by 25%.

1 Introduction

A number of studies have demonstrated the benefits of web prefetching [12, 17, 24, 25, 32, 33, 42, 52]. And the attractiveness of prefetching appears likely to rise in the future as the falling prices of disk storage [14] and network bandwidth [41] make it increasingly attractive to trade increased consumption of these resources to improve response time and availability and thus reduce human wait time [7].

Despite these benefits, prefetching systems have not been widely deployed because of two concerns: interference and deployability. First, if a prefetching system is too aggressive, it may interfere with demand requests to the same service (self-interference) or to other services (cross-interference) and hurt overall system performance. Such interference may occur at the server, in the communication network or at the client. Second, if a system requires modifications to the existing HTTP protocol [19] , it may be impractical to deploy. The large number of deployed clients and networks in the Internet makes it difficult to change clients, and the increasing complexity of servers [23, 26, 28, 46, 55] makes it difficult to change servers. What we therefore need is a prefetching system that (a) avoids interference at clients, networks, and servers and (b) does not require changes to the HTTP protocol and the existing infrastructure (client browsers, networks and servers).

In this paper, we make three contributions. First, we present NPS, a novel non-interfering prefetching system for the web that – (1) avoids interference by effectively utilizing only spare resources on the servers and the network and (2) is deployable with no modifications to the HTTP protocol and existing infrastructure. To avoid interference at the server, NPS monitors the server load externally and restricts the prefetch load imposed on it accordingly. To avoid interference in the underlying network, NPS uses TCP-Nice for low-priority network transfers [51]. Finally, it uses a set of heuristics to control resource usage at the client. To work with existing infrastructure, NPS modifies HTML pages to include JavaScript^{TM} code to issue prefetch requests, and wraps the server infrastructure with simple external modules that require no knowledge of, or no modifications to the internals of existing servers. Our measurements of the prototype under real web load trace indicate that NPS is both non-interfering and efficient under different network and server load conditions. For example, in our experiments on a heavily loaded network with little spare capacity, we observe that a threshold-based prefetching scheme causes response times to increase by a factor of 7 due to interference, whereas prefetching using NPS contains this increase to less than 30%.

Second, and on a broader note, we propose a self-tuning architecture for prefetching that eliminates the need for traditional "threshold" magic numbers that are typically used to limit the interference that prefetching inflicts on demand requests. This architecture divides prefetching into two separates tasks – (i) prediction and (ii) resource management. The predictor proposes prioritized lists of high-valued documents to prefetch. The resource manager limits the number of documents to prefetch and schedules the prefetch requests to avoid interference with demand requests and other applications. This separation of concerns has three advantages – (i) it simplifies the design and deployment of prefetching systems by eliminating the need to choose appropriate thresholds for an environment and update them with changing conditions, (ii) it reduces the risk of interference caused by prefetching that relies on manually set thresholds, especially during periods of unanticipated high load, (iii) it increases the benefits of prefetching by prefetching more aggressively than would otherwise be safe during periods of low or moderate load. We believe that these advantages would also apply to prefetching systems in many environments beyond the web.

Third, we explore the design space for building a web prefetching system, given the requirement of avoiding or minimizing changes to existing infrastructure. We find that it is straightforward to deploy prefetching that ignores the problem of interference, and it is not much more difficult to augment such a system to avoid server interference. Extending the system to also avoid network interference is more involved, but doing so appears feasible even under the constraint of not modifying current infrastructure. Unfortunately, we were unable to devise a method to completely eliminate prefetching's interference at existing clients: in our system prefetched data may displace more valuable data in a client cache. It appears that a complete solution may eventually require modifications at the client [6, 8, 44]. For now, we develop simple heuristics that reduce this interference.

The rest of the paper is organized as follows. Section 2 discusses the requirements and architecture of a prefetching system. Sections 3, 4 and 5 present the building blocks for reducing interference at servers, networks and clients. Section 6 presents the prefetch mechanisms that we develop to realize the prefetching architecture. Section 7 discusses the details of our prototype and evaluation. Section 8 presents some related work and section 9 concludes.

2 Requirements and Alternatives

There appears to be a consensus among researchers on a high level architecture for prefetching in which a server sends a list of objects to a client and the client issues prefetch requests for the objects on the list [9, 36, 42] . This division of labor allows servers to use global object access patterns and service-specific knowledge to determine what should be prefetched, and it allows clients to filter requests through their caches to avoid repeatedly fetching objects. In this paper, we develop a framework for prefetching that follows this organization and that seeks to meet two other important requirements: self tuning resource management and deployability without modifying existing protocols, clients, proxies, or servers.

2.1 Resource Management

Services that prefetch should balance the benefits against the risk of interference. Interference can take the form of *self-interference*, where a prefetching service hurts its own performance by interfering with its demand requests, and *cross-interference*, where the service hurts the performance of other applications on the prefetching client, other clients, or both.

Limiting interference is essential because many prefetching services have potentially unlimited bandwidth demand, where incrementally more bandwidth consumption provides incrementally better service. For example, a prefetching system can improve hit rate and hence response times by fetching objects from a virtually unlimited collection of objects that have non-zero probabilities of access [5, 8], or by updating cached copies more frequently [10, 50, 52].

Interference can occur at any of the critical resources in the system.

- **Server:** Prefetching consumes extra resources on the server such as processing time, memory space and disk.
- **Network:** Prefetching causes extra data packets to be transmitted over the network, potentially increasing queuing delays and packet drops.
- **Client:** Prefetching results in extra processing at clients. Furthermore, aggressive prefetching can pollute a browser's memory and disk caches.

A common way of achieving balance between the benefits and costs of prefetching is to select a threshold and prefetch objects whose estimated probability of use before modification or eviction from the cache exceeds that threshold [17, 29, 42, 52]. There are at least two problems with such "magic number"-based approaches. First, it is difficult for even an expert to set thresholds to optimum values to balance costs and benefits—although thresholds relate closely to the benefits of prefetching, they have little obvious relationship to the costs of prefetching [7, 21]. Second, appropriate thresholds to balance costs and benefits may vary over time as client, network, and server load conditions change over seconds (e.g., changing workloads or network congestion [56]), hours (e.g., diurnal patterns), and months (e.g., technology trends [7, 41]).

Our goal is to construct a self-tuning resource module that prevents prefetch requests from interfering with demand requests. Such an architecture will simplify the design of prefetching systems by separating the tasks of prediction and resource management. Prediction algorithms may specify arbitrarily long lists of the most beneficial objects to prefetch sorted by benefit, and the resource management module issues requests for these objects and ensures that these requests do not interfere with demand requests or other system activities. In addition to simplifying system design, such an architecture could have two performance advantages over statically set prefetch thresholds. First, such a system can reduce interference – when resources are scarce, it would reduce prefetching aggressiveness. Second, such a system may increase the benefits of prefetching when resources are plentiful by allowing more aggressive prefetching than would otherwise be considered safe.

2.2 Deployability

Many proposed prefetching mechanisms suggest modifying the HTTP/1.1 protocol [4, 15, 17, 42], to create a new request type for prefetching. An advantage of extending the protocol is that clients, proxies, and servers could then distinguish prefetch requests from demand requests and potentially schedule them separately to prevent prefetch requests from interfering with demand requests [15]. However, such mechanisms are not easily deployable because modifying the protocol implies modifying the widely-deployed infrastructure that supports the current protocol including existing clients, proxies, and servers. As web servers evolve and increase in their complexity, requests may traverse not only a highly optimized web server [43, 49, 54, 55] but also a number of other complex modules such as commercial databases, application servers or virtual machines for assembling dynamic content (e.g., Apache tomcat for executing Java Servlets and JavaServer pages), distributed cluster services [2, 23], and content delivery networks. Modifying servers to separate prefetch requests from demand requests maybe complex or infeasible under such circumstances.

If interference were not a concern, a simple prefetching system could easily be built with the present infrastructure, where clients can be made to prefetch without any modifications to the protocol. For example, servers can embed JavaScript code or a Java applet [20], to fetch specified objects over the network and load them into the browser cache. An alternative way is to add invisible frames to the demand content that include and thereby preload the prefetch content.

In this paper, we adapt such techniques to avoid interference while maintaining deployability.

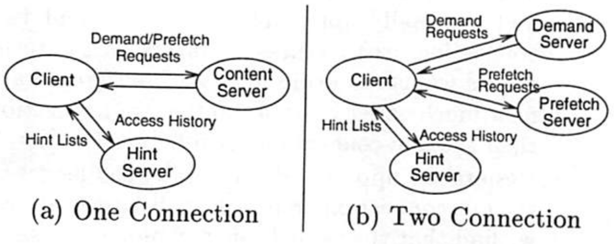
2.3 Architectural Alternatives

In this subsection, we present an overview of two alternative architectures to build a prefetching system. The high-level description in this section is intended only to provide a framework for discussing resource management strategies at the server, network, and client in sections 3 through 5. These architectures and resource management strategies are pertinent regardless of whether prefetching is implemented using a new protocol or by exploiting existing infrastructure. In Section 6, we describe how our implementation realizes one of these architectures in an easily deployable way.

We begin by making the following assumptions about client browsers:

- For easy deployability of the prefetching system, browsers should be unmodified.
- Browsers match requests to documents in their caches based on (among other parameters) the server name and the file name of the object on the server. Thus files of the same name served from different servers are considered to be different.
- Browsers may multiplex multiple client requests to a given server on one or more persistent connections [19].

Figure 1 illustrates what we call the one-connection and two-connection architectures respectively. In both architectures, clients send their access histories to the *hint server* and get a list of documents to prefetch. The hint server uses either online or offline prediction algorithms to compute the hint lists consisting of the most probable documents that the users might request in the future.



(a) One Connection (b) Two Connection

Figure 1. Design Alternatives for a Prefetching System

2.3.1 One Connection

In the one connection architecture (Figure 1(a)), a client fetches both demand and prefetch requests from the same content server. Since browsers multiplex requests over established connections to servers, and since browsers do not differentiate between demand and prefetch requests, each TCP connection may interleave prefetch and demand requests and responses.

Sharing connections can cause prefetch requests to interfere with demand requests for network and server resources. If interference can be avoided, this system is easily deployable. In particular, objects fetched from the same server share the domain name of the server. So, unmodified client browsers can use cached prefetched objects to service demand requests.

2.3.2 Two Connection

In the two connection architecture(Figure 1(b)), a client fetches demand and prefetch requests from different servers or from different ports on the same server. This architecture thus segregates demand and prefetch requests on separate network connections.

Although the two connection architecture simplifies the mechanisms for reducing interference at the server by segregation, this solution appears to complicate the deployability of the system. Objects with the same names fetched from different servers are considered different by the browsers. So, browsers can not directly use the prefetched objects to service demand requests.

2.3.3 Comparison

In the following sections, we show how to address the limitations of both architectures.

- Some of the techniques we develop for avoiding interference are useful for the one connection architecture, but some are less so. In particular, our strategy for reducing interference at servers is based on end-to-end performance and is equally applicable to the one and two connection architectures. Conversely, the techniques we use to avoid network interference appear much easier to apply to the two-connection than the one-connection architecture.
- Despite the apparent deployability challenges to the two connection architecture discussed above, we find that the same basic technique we use to make unmodified browsers prefetch data for the one connection architecture can be adapted to support the two connection architecture as well.

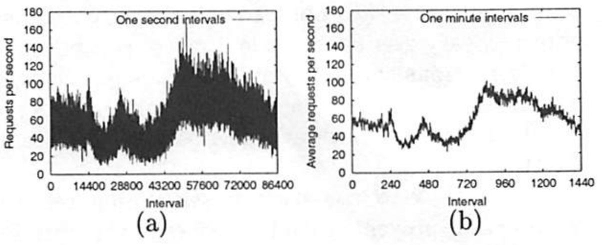


Figure 2. Server loads averaged over (a) 1-second and (b) 1-minute intervals for the IBM sporting event workload.

We conclude that both architectures are tenable in some circumstances. If server load is the primary concern and if network load is known not to be a major issue, then the one connection prototype may be simpler than the two connection prototype. At the same time, the two connection prototype is feasible and deployable and manages both network and server interference. Given that networks are a globally shared resource, we recommend the use of two connection architecture in most circumstances.

3 Server Interference

An ideal system for avoiding server interference would cause no delay to demand requests in the system and utilize significant amounts of any spare resources on servers for prefetching. Such a system needs to cope with, and take advantage of, changing workload patterns over various time scales. HTTP request traffic arriving at a server often is bursty with the burstiness being observable at several scales of observation [13] and with peak rates exceeding the average rate by factors of 8 to 10 [37]. For example, Figure 2 shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals. It is crucial for the prefetching system to be responsive to such bursts to balance utilization and risk of interference.

3.1 Alternatives

There are a variety of ways to prevent prefetch requests from interfering with demand requests at servers.

Local scheduling Server scheduling can help use the spare capacity of existing infrastructure for prefetching in a non-interfering manner. In principle, existing schedulers for processor, memory [29, 31, 44], and disk [35] could prevent low-priority prefetch requests from interfering with high-priority demand requests. Furthermore, as these schedulers are intimately tied to the operating system, they should be highly efficient in delivering whatever spare capacity exists to prefetch requests even over fine time scales. Note that local scheduling is equally applicable to both one- and two-connection architectures.

For many services, however, server scheduling may not be easily deployable for two reasons. First, although several modern operating systems support process schedulers that can provide strict priority scheduling, few provide memory, cache or disk schedulers that isolate prefetch requests from demand requests. Second, even if an operating system provides the needed support, existing servers would have to be modified to differentiate between prefetch and demand requests with scheduling priorities as they are serviced [3]. This second requirement appears particularly challenging given the increasing complexity of servers, in which requests may traverse not only a highly-tuned web server [43, 49, 54, 55] but also a number of other complex modules such as commercial databases, application servers or virtual machines for assembling dynamic content (e.g., Apache tomcat for executing Java Servlets and JavaServer pages), distributed cluster services [2, 23], and content delivery networks.

Separate prefetch infrastructure An intuitively simple way of avoiding server interference is to use separate servers to achieve complete isolation of prefetch and demand requests. In addition to the obvious strategy of providing separate demand and prefetch machines in a centralized cluster, a natural use of this strategy might be for a third-party "prefetch distribution network" to supply geographically distributed prefetch servers in a manner analogous to existing content distribution networks. Note that this alternative is not available to the one-connection architecture.

However, separate infrastructure needs extra hardware and hence may not be an economically viable solution for many web sites.

End-to-end monitoring A technique based on end-to-end monitoring estimates the overall load (or spare capacity) on the server by periodically probing the server with representative requests and measuring the response times of the replies. Low response times indicate that the server has spare capacity and high response times indicate that the server is loaded. Based on such an estimate, the monitor utilizes the spare capacity on the server by controlling the number and aggressiveness of prefetching clients.

An advantage of end-to-end monitoring is that it requires no modifications to existing servers. Furthermore, it can be used by both one- and two- connection prefetching architectures. The disadvantage of such an approach is that its scheduling precision is likely to be less than that of a local scheduler that has access to the internal state of the server and operating system. Moreover, an end-to-end monitor may not be responsive enough to bursts in load over fine time scales.

In the following subsections, we discuss issues involved in designing an end-to-end monitor in greater detail, present our simple monitor design, and evaluate its efficacy in comparison to server scheduling.

3.2 End-to-end Monitor Design

Figure 3 illustrates the architecture of our monitor-controlled prefetching system. The monitor estimates the server's spare capacity and sets a *budget* of prefetch requests permitted for an interval. The hint server adjusts the load imposed by prefetching on the server by ensuring that the sum across the hint lists returned to clients does not exceed the budget. Our monitor design must adress two issues: (i) budget estimation and (ii) budget distribution across clients.

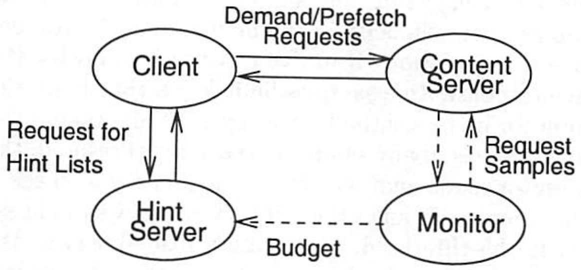


Figure 3. A Monitored Prefetching System

Budget estimation The monitor periodically probes the server with HTTP requests to representative objects and measures the response times. The monitor increases the budget when the response times are below the objects' *threshold* values and decreases the budget otherwise.

As probing is an intrusive technique, choosing an appropriate rate of probing is a challenge. A high rate makes the monitor more reactive to load on the server, but also adds extra load on the server. On the other hand, a low rate makes the monitor react slowly, and can potentially lead to interference to the demand requests. Similarly, the exact policy for increasing and decreasing the budget must balance the risk of causing interference against underutilization of spare capacity.

Budget distribution The goal of this task is to distribute the budget among the clients such that (i) the load due to prefetching on the server is contained within the budget for that epoch and is distributed uniformly over the interval, (ii) a significant fraction of the budget is utilized over the interval, and (iii) clients are responsive to changing load patterns at the server. The two knobs that the hint server can manipulate to achieve these goals are (i) the size of the hint list returned to the clients and (ii) the subset of clients that are given permission to prefetch. This flexibility provides a freedom to choose from many policies.

3.3 Monitor Prototype

Our prototype uses simple, minimally tuned policies for budget estimation and budget distribution. Future work may improve the performance of our monitor.

The monitor probes the server in epochs, each approximately 100 ms long. In each epoch, the monitor collects a response time sample for a representative request. In the interest of being conservative – choosing non-interference even at the potential cost of reduced utilization – we use an additive increase(increase by 1), multiplicative decrease (reduce by half) policy. AIMD is commonly used in network congestion control [30] to conservatively estimate spare capacity in the network and be responsive to congestion. If in five consecutive epochs, the five response time samples lie below a threshold, the monitor increases the budget by 1. While taking the five samples, if any sample exceeds the threshold, the monitor sends another probe immediately to check if the sample was an outlier. If even the new sample exceeds the threshold, indicating a loaded server, the monitor decreases the budget by half and restarts collecting the next five samples.

In our simple prototype, we manually supply the representative objects's threshold response times. However, it is straightforward because of the predictable pattern in which response times vary with load on server systems – a nearly constant value of response time for low load followed by a sharp rise beyond the "knee" for high load. As part of our future work, we intend to make the monitor automatically pick thresholds in a self-tuning manner.

The hint server distributes the current budget among client requests that arrive in that epoch. We choose to set the hint list size to the size of one document (a document corresponds to a HTML page and all embedded objects). Our policy lets clients to return quickly for more hints and thus be more responsive to changing load patterns on the server. Note that returning larger hint lists would reduce the load on the hint server, but it would reduce the system's responsiveness and its ability to avoid interference. We control the number of simultaneously prefetching clients, and thus the load on the server, by returning to some clients a hint list of zero size and a directive to wait until the next epoch to fetch the next hint list. For example, if B denotes the budget in the current epoch, and N the expected number of clients in that epoch, D the number of files in a document, and τ the epoch length, the hint server accepts a fraction $p = min(1, \frac{B \cdot \tau}{N \cdot D})$ of requests to prefetch on part of clients in that epoch and returns hintlists of zero length for other requests. Note that other designs are possible. For example, the monitor can integrate with the prefetch prediction algorithm to favor prefetching by clients for which the predictor can identify high-probability items and defer prefetching by clients for which the predictor identifies few high-value targets.

Since the hint server does not a priori know the number of client requests that will come in an epoch, it estimates that value with the number of requests that come in the previous epoch. If more than the estimated number of requests arrive in a epoch, the hint server replies with list of size zero and a directive to retry in the next epoch to those extra requests. If fewer clients arrive, some of the budget can get wasted. However, in the interest of avoiding interference, we choose to allow such wastage of budget.

In the following Section 3.3.1, we evaluate the performance of our prototype with respect to the goals of reducing interference and reaping significant spare bandwidth and compare it with the other resource management alternatives.

3.3.1 Evaluation

In evaluating resource management algorithms, we are mainly concerned with interference that prefetching could cause and less with the benefits obtained. We therefore abstract away prediction policies used by services by prefetching sets of dummy data from arbitrary URLs at the server. The goal of the experiments is to compare the effectiveness of different resource management alternatives in avoiding server interference against the ideal case (when no prefetching is done) with respect to the following metrics: (i) *cost:* the amount of interference in terms of demand response times and (ii) *benefit:* the prefetch bandwidth.

We consider the following resource management algorithms for this set of experiments:

1. No-Prefetching: Ideal case, when no prefetching is done or when we use a separate prefetching infrastructure.
2. No-Avoidance: Prefetching with no interference avoidance with fixed aggressiveness. We set the aggressiveness by setting *pfrate*, which is the number of documents prefetched for each demand document. For a given service, a given prefetch threshold will correspond to some average *pfrate*. We use fixed *pfrate* values of 1 and 5.

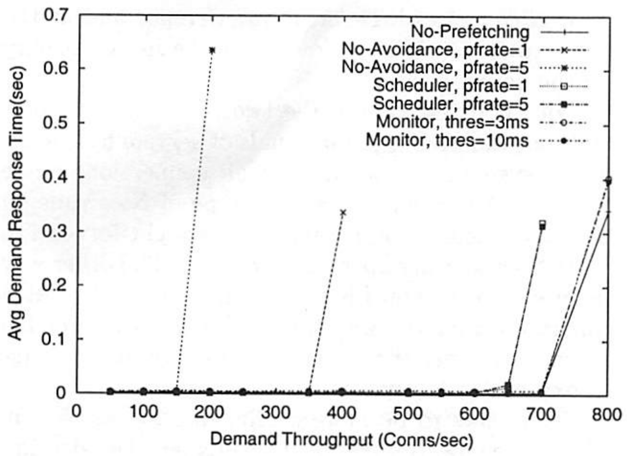


Figure 4. Effect of prefetching on demand throughput and response times with various resource management policies

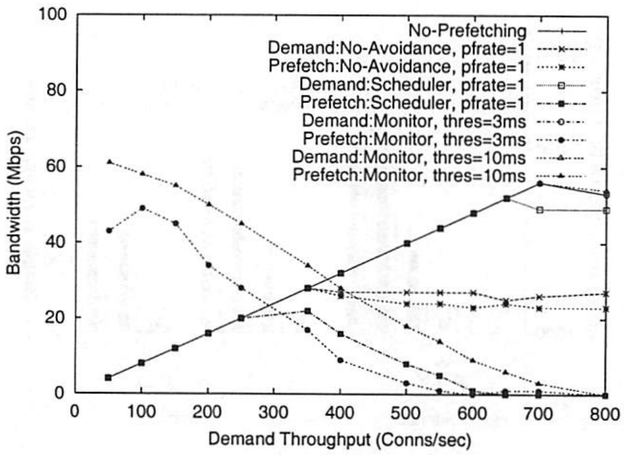


Figure 5. Prefetch and demand bandwidths achieved by various algorithms

3. Scheduler: As a simple local server scheduling policy, we choose *nice*, the process scheduling utility in Unix. We again use fixed *pfrate* values of 1 and 5. This simple server scheduling algorithm is only intended as a comparison; more sophisticated local schedulers may better approximate the ideal case.
4. Monitor: We perform experiments for two threshold values of 3ms and 10ms.

For evaluating algorithms 2 and 4, we set up one server serving both demand and prefetch requests. These algorithms are applicable in both one connection and two connection architectures. Our prototype implementation of algorithm 3 requires that the demand and prefetch requests be serviced by different processes and thus is applicable only to the two connection architecture. We use two different servers listening on two ports on the same machine, with one server run at a lower priority using the Linux *nice*. Note that the general local scheduling approach is equally applicable to the one-connection architecture with more intrusive server modifications.

Our experimental setup includes Apache HTTP server [1] running on a 450MHz Pentium II, with 128MB of memory. To generate the client load, we use httperf [38] running on four different Pentium III 930MHz machines. All machines run the Linux operating system.

We use two workloads in our experiments. Our first workload generates demand requests to the server at a constant rate. The second workload is a one hour subset of the IBM sporting event server trace, whose characteristics are shown in Figure 2. We scale up the trace in time by a factor of two, so that requests are generated at twice the original rate, as the original trace barely loads our server.

Constant workload Figure 4 shows the demand response times with varying demand request arrival rate. The graph shows that both Monitor and Scheduler algorithms closely approximate the behavior of No-Prefetching in not affecting the demand response times. Whereas, the No-Avoidance algorithm with fixed *pfrate* values significantly damages both the demand response times and the maximum demand throughput.

Figure 5 shows the bandwidth achieved by the prefetch requests and their effect on the demand bandwidth. The figure shows that No-Avoidance adversely affects the demand bandwidth. Conversely, both Scheduler and Monitor reap spare bandwidth for prefetching without much decrease in the demand bandwidth. Further, at low demand loads, a fixed pfrate prevents No-Avoidance from utilizing the full available spare bandwidth. The problem of too little prefetching when demand load is low and too much prefetching when demand load is high illustrates the problem with existing threshold strategies. As hoped, the Monitor tunes prefetch aggressiveness of the clients such that essentially all of the spare bandwidth is utilized.

IBM server trace In this set of experiments, we compare the performance of the four algorithms for the IBM server trace. Figure 6 shows the demand response times and prefetch bandwidth in each case. The graph shows that the No-Avoidance case affects the demand response times significantly as *pfrate* increases. The Scheduler and Monitor cases have less adverse effects on the demand response times.

These experiments show that resource management is an important component of a prefetching system because overly aggressive prefetching can significantly hurt demand response time and throughput while timid prefetching gives up significant band-

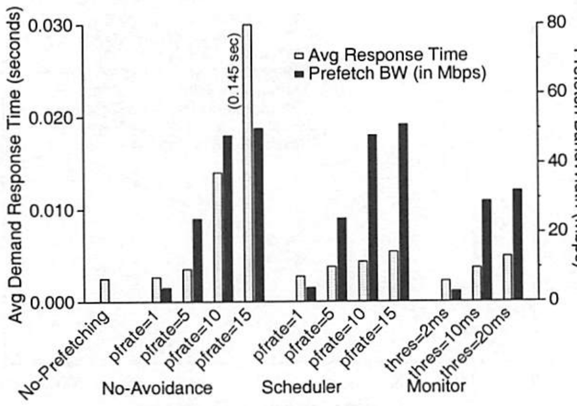


Figure 6. Performance of No-Avoidance, Scheduler and Monitor schemes on the IBM server trace

width. They also illustrate a key problem with constant non-adaptive magic numbers in prefetching such as the threshold approach that is commonly proposed. The experiments also provide evidence of the effectiveness of the monitor in tuning prefetch aggressiveness of clients to reap significant spare bandwidth while keeping interference at a minimum.

4 Network Interference

Mechanisms to reduce network interference could, in principle, be deployed at clients, intermediate routers, or servers. For example, clients can reduce the rate at which they receive data from the servers using TCP flow control mechanisms [48]. However, it is not clear how to set the parameters to such mechanisms or how to deploy them given existing infrastructure. Prioritization in routers that provide differentiated service to prefetch and demand packets can avoid interference effectively [47]. However, router prioritization is not easily deployable in the near future. We focus on server based control because of the relative ease of deployability of server based mechanisms and their effectiveness in avoiding both self- and cross-interference.

In particular, we use a transport level solution at the server – TCP-Nice [51]. TCP-Nice is a congestion control mechanism at the sender that is specifically designed to support background data transfers like prefetching. Background connections using Nice operate by utilizing only spare bandwidth in the network. They react more sensitively to congestion and backoff when a possibility of congestion is detected, giving way to foreground connections. In our previous study [51], we provably bound the network interference caused by Nice under a simple network model. Furthermore, our experimental evidence under wide range of conditions and workloads shows that Nice causes little or no interference and at the same time reaps a large fraction of the spare capacity in the network.

Nice is deployable in the two connection context without modifying the internals of servers by configuring systems to use Nice for all connections made to the prefetch server. A prototype of Nice runs on Linux currently, and it should be straight-forward to port Nice to other operating systems. The other way to use Nice in non-Linux environments is to put a Linux machine running Nice in front of the prefetch server and make the Linux machine serve as a reverse proxy or a gateway.

It appears to be more challenging to use Nice in the one connection case. In principle, the Nice implementation allows flipping a connection's congestion control algorithm between standard TCP (when serving demand requests) and Nice (when serving prefetch requests). However, using this approach for prefetching faces a number of challenges: (1) Flipping modes causes packets already queued in the TCP socket buffer to inherit the new mode. Thus, demand packets queued in the socket buffer may be sent at low-priority while prefetch packets may be sent at normal-priority, thus causing network interference. Ensuring that demand and prefetch packets are sent in the appropriate modes would require an extension to Nice and a fine-grained coordination between the application and the congestion control implementation. (2) Nice is designed for long network flows. It is not clear if flipping back and forth between congestion control algorithms will still avoid interference and gain significant spare bandwidth. (3) HTTP/1.1 pipelining requires replies to be sent in the order requests were received, so demand requests may be queued behind prefetch requests, causing demand requests to perceive increased latencies. One way to avoid such interference may be to quash all the prefetch requests queued in front of the demand request. For example, we could send a small error message (eg. HTTP response code 307 – "Temporary Redirect" with a redirection to the original URL) as a response to the quashed prefetch requests.

Based on these challenges, it appears simpler to use the two connection architecture when the network is a potential bottleneck. A topic for future work is to explore these challenges and determine if a deployable one connection architecture that avoids network interference can be devised.

5 Client Interference

Prefetching may interfere with the performance of a client in at least two ways. First, prefetch requests consume processing cycles and may, for instance, delay rendering of demand pages. Second, prefetched data may displace demand data from the cache and thus hurt demand hit rates for the prefetching service or other services.

As with the interference at the server discussed above, interference between client processes could, in principle, be addressed by modifying the client browser (and, perhaps, the client operating system) to use a local processor scheduler to ensure that processing of prefetch requests never interferes with processing of demand requests. Lacking that option, we resort to a simpler approach: as described in Section 6, we structure our prefetch mechanism to ensure that processing prefetch requests does not begin until after the loading and rendering of the demand page, including all inline images and recursive frames. Although this approach will not help reduce cross-interference with other applications at the client, it may avoid a potentially common case of self-interference of the prefetches triggered by a page delaying the rendering of that page.

Similarly, a number of storage scheduling algorithms exist that balance caching prefetched data against caching demand data [6, 8, 31, 44]. Unfortunately, all of these algorithms require modifications to the cache replacement algorithm.

Because we assume that the client cannot be modified, we resort to two heuristics to limit cache pollution caused by prefetching. First, in our system, services place a limit on the ratio of prefetched bytes to demand bytes sent to a client. Second, services can set the `Expires` HTTP header to a value in the relatively near future (e.g., one day in the future) to encourage clients to evict prefetched document earlier than they may otherwise have done. These heuristics have an obvious disadvantage: they resort to magic numbers similar to those in current use, and they suffer from the same potential problems: if the magic numbers are too aggressive, prefetching services will interfere with other services, and if they are too timid, prefetching services will not gain the benefits they might otherwise gain. Fortunately, there is reason to hope that performance will not be too sensitive to this parameter. First, disks are large and growing larger at about 100% per year [14] and relatively modest-sized disks are effectively infinite for many client web cache workloads [52]. So, disk caches may absorb relatively large amounts of prefetch data with little interference. Second, hit rates fall relatively slowly as disk capacities shrink [5, 52], which would suggest that relatively large amounts of polluting prefetch data will have relatively small effects on demand hit rate.

Figure 7 illustrates the extent to which our heuristics can limit the interference of prefetching on hit rates. We use the 28-day UCB trace of 8000 unique clients from 1996 [22] and simulate the hit rates of

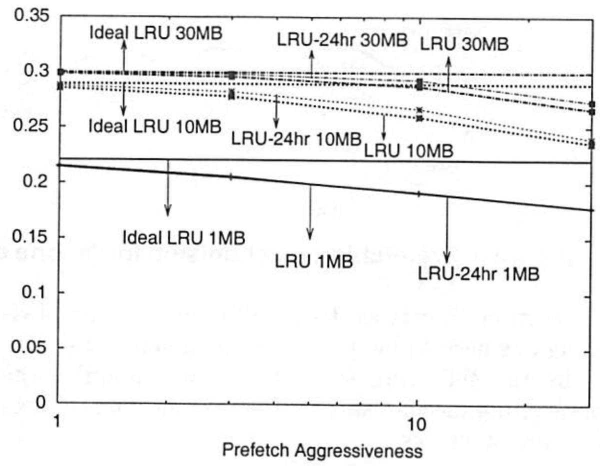


Figure 7. Effect of prefetching on demand hit rate

1 MB, 10 MB and 30 MB per-client caches. Note that these cache sizes are small given, for example, Internet Explorer's defaults of using 3% of a disk's capacity (e.g., 300 MB of a 10 GB disk) for web caching. On the x-axis, we vary the number of bytes of dummy prefetch data per byte of demand data that are fetched after each demand request. In this experiment, 20% of services use prefetching at the specified aggressiveness and the remainder do not, and we plot the demand hit rate of the *non*-prefetching services. Ideally, these hit rates should be unaffected by prefetching. As the graph shows, hit rates fall gradually as prefetching increases, and the effect shrinks as cache sizes get larger. For example, if a client cache is 30 MB and 20% of services prefetch aggressively enough that each prefetches ten times as much prefetch data as the client references demand data, demand hit rates fall from 29.9% to 28.7%.

6 Prefetching Mechanism

Figure 8 illustrates the key components of the one and two connection architectures. The one-connection mechanism consists of an unmodified client, a content server that serves both demand and prefetch requests, a munger that modifies content on the content server to activate prefetching and a hint server that gives out hint lists to the client to prefetch. The hint server also includes a monitor that probes the content server and estimates the spare capacity at the server and accordingly controls the number of prefetching clients.

The two-connection prototype, along with the components above, also consists of a prefetch server that is a copy of the demand server (running either on a separate machine or on a different port on the same machine) and a front-end that intercepts certain requests to the demand server and returns appropriate

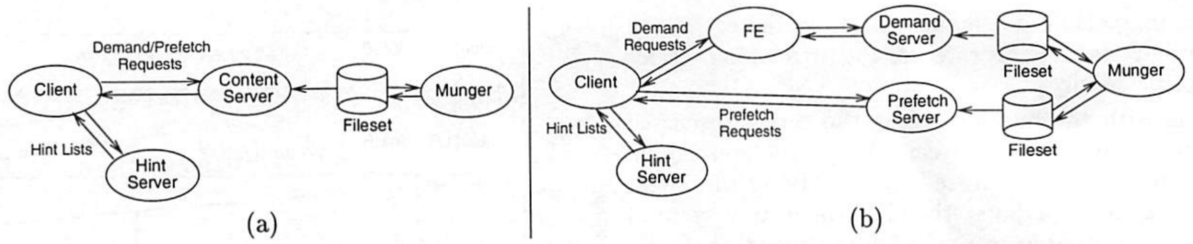


Figure 8. Prefetching mechanisms for (a) one connection and (b) two connection architectures.

redirection objects as described later, thereby obviating any need to modify the original demand server.

In the following subsections, we describe the prefetching mechanisms for the one and two connection architectures.

6.1 One-connection Prefetching Mechanism

Content modification The munger augments each HTML document with pieces of JavaScript and HTML code that cause the client to prefetch.

On demand fetch

1. Client requests an augmented HTML document.
2. When an augmented HTML document (Figure 9) finishes loading into the browser, the `pageOnLoad()` function is called. This function calls `getPfList()`, a function defined in `pfalways.html` (Figure 10). The file `pfalways.html` is loaded within every augmented HTML document. `pfalways.html` is cacheable and hence does not need to be fetched everytime a document gets loaded.
3. `getPfList()` sends a request for `pflist.html` to the hint server with the name of the enclosing document, the name of the previous document in history (the enclosing document's referer) and TURN=1 as extra information embedded in the URL.
4. The hint server receives the request for `pflist.html`. Since the client fetches a `pflist.html` for each HTML document (even if the HTML document is found in the cache), the client provides the hint server with a history of accesses to aid in predicting hint lists. In Figure 10, PFCOOKIE contains the present access (`document.referrer`) and the last access (`prevref`) by the client. The hint updates the history and predicts a list of documents to be prefetched by the client based on that client's history and the global access patterns. It puts these predictions into the response `pflist.html` such as shown in 11, which it returns to the client.
5. `pflist.html` replaces `pfalways.html` on the client. After `pflist.html` loads, the `preload()` function in its body preloads the documents to be prefetched from the prefetch server (which is same as the demand server in the one connection case).
6. After all the prefetch documents are preloaded, the `myOnLoad()` function calls `getMore()` that replaces the current `pflist.html` by fetching a new version with TURN=TURN+1.

```
<HTML> <HEAD> <! -- existing header goes here -- >
<SCRIPT LANGUAGE="JavaScript">
function pageOnLoad() {
 myiframe.getPFlist(document.referrer);
} </SCRIPT> </HEAD> <BODY>

<! -- existing body goes here -- >
if(null == window.onload) {
 window.onload = pageOnLoad();}
else {
 var origfn = window.onload;
 window.onload = function(){origfn();pageOnLoad();};}

<IFRAME SRC="pfalways.html" name="myiframe"
        width=0 height=0 frameborder=0>
</IFRAME> </BODY> </HTML>
```

Figure 9. Augmentation of HTML pages

```
<HTML> <HEAD> <SCRIPT LANGUAGE="JavaScript">
function getPFList(var prevref) {
 document.location="HINT-SERVER/pflist.html+PCOOKIE="
       + document.referrer + "+" + prevref + TURN=1;
 document.close();
} </SCRIPT> </HEAD> </HTML>
```

Figure 10. pfalways.html

Steps 5 and 6 repeat until the hint server has sent everything it wants, at which point the hint server returns a `pflist.html` with no `getMore()` call. When there is not enough budget left at the server, the hint server sends a `pflist.html` with no files to prefetch and a delay, after which the `getMore()` function gets called. The information TURN breaks the (possibly) long list of prefetch suggestions into a "chain" of short lists.

On demand fetch of a prefetched document The client browser fetches it from the cache as if it is a cache hit.

```
<HTML> <HEAD> <SCRIPT LANGUAGE="JavaScript">

function myOnLoad() { //exeutes after body loads
 preload("DEMAND-SERVER/c.html"); //For two-conn only
 getMore() ;
}
function getMore() {
 document.location="HINT-SERVER/pflist.html +
                   PCOOKIE=" + document.referrer +
                   "+" + prevref + "+" + "TURN=2";
 document.close();
}

var myfiles=new Array()
function preload(){
 for (i=0;i<preload.arguments.length;i++){
  myfiles[i]=new Image() ;
  myfiles[i].src=preload.arguments[i] ;
 }
} </SCRIPT> </HEAD>

<BODY onload="myOnLoad()">
<SCRIPT LANGUAGE="JavaScript">
preload("PREFETCH-SERVER/a.jpg",
        "PREFETCH-SERVER/b.jpg",
        "PREFETCH-SERVER/c.html");
</SCRIPT> </BODY> </HTML>
```

Figure 11. An example pflist.html returned by the hint server

```
<HTML> <SCRIPT LANGUAGE="JavaScript">
if (document.referrer.indexOf ("pflist") < 0)
         document.location="PREFETCH-SERVER/c.html";
document.close();
</SCRIPT> </HTML>
```

Figure 12. Wrapper for c.html, stored in cache as DEMAND-SERVER/c.html

6.2 Two-connection Prefetching Mechanism

The two-connection prototype employs the same basic mechanism for prefetching as the one-connection prototype. However, since browsers identify cached documents using both the server name and document name, documents fetched from prefetch server are not directly usable to serve demand requests. In order to fix this problem, we modify step 6 such that before calling `getMore()`,

6.a The `myOnLoad()` function (Figure 11) requests a *wrapper* (redirection object) from the demand server for the document that was prefetched.

6.b The frontend intercepts the request (based on the referer field) and responds with the wrapper (Figure 12) that loads the prefetched document in response to a client's demand request.

The prefetch server serves a modified copy of the content on the demand server. Note that the relative links in a webpage on the demand server point to pages on demand server. Hence, all relative links in the prefetch server's content are changed to absolute links, such that when client clicks on a link in the prefethed web page, the request is sent to the demand server. Also, all absolute links to inline objects in the page are changed to be absolute links to the prefetch server, so that prefetched inline objects are used. Since prefetch and demand servers are considered as different domains by the client browser, JavaScript security models [40] prevent scripts in prefetched documents to access private information of the demand documents and vice versa. However, to fix this problem, JavaScript allows us to explicitly set the `document.domain` property of each HTML document to a common suffix of prefetch and demand servers. For example, for servers `demand.cs.utexas.edu` and `prefetch.cs.utexas.edu`, all the HTML documents can set their `document.domain` property to `cs.utexas.edu`.

On demand fetch of a prefetched document: (i) a hit results for the wrapper in the cache, (ii) at the loading time, the wrapper replaces itself with the prefetched document from the cache, (iii) inline objects in the prefetched document point to objects from the prefetch server and hence are found in the cache as well, and (iv) links in the prefetched document point to the demand server.

This mechanism has two limitations. First, prefetched objects might get evicted from the cache before their wrappers. In such a case, when the wrapper loads for a demand request, a new request will be sent to the prefetch server. Since sending a request to the prefetch server in response to a demand request could cause undesirable delay, we reduce such occurrences by setting the expiration time of the wrapper to a value smaller than the expiration of the prefetched object itself. Second, but not a significant limitation is that some objects may be fetched twice, once as demand and once as prefetch objects as the browser cache considers them as different objects.

6.3 Prediction

For our experiments, we use prediction by partial matching [11] (PPM-n/w) to generate hint lists for prefetching. The algorithm uses a client's n most recent requests to the server for non-image data to predict URLs that will appear during a subsequent window that ends after the w'th non-image request to the server. Our prototype uses n=2 and w=10.

In general, the hint server can be made to use any prediction algorithm. It can be made to use standard algorithms proposed in the literature [17, 18, 24, 42] or others that utilize more service specific information such as a news site that prefetches stories relating to topics that interest a given user.

6.4 Alternatives

We explored other alternatives for prefetching in the two-connection architecture. We could have used a Java Applet instead of the JavaScript in Figure 9. One could also use a zero-pixel frame that loads the prefetched objects instead of JavaScript. The refresh header in HTTP/1.1 could be exploited to iteratively prefetch a list of objects by setting the refresh time to a small value.

As an alternative to using wrappers, we also considered maintaining state explicitly at the client to store information about whether a document has already been prefetched. Content could be augmented with a script to execute on a hyperlink's `onClick` event that checks this state information before requesting a document from the demand server or prefetch server. Similar augmentation could be done for inline objects. Tricks to maintain state on the client can be found in [45].

7 Prototype and Evaluation

Our prototype uses the two connection architecture whose prefetching mechanism is shown in Figure 8(b). We use Apache 2.0.39 as the server, hosted on a 450MHz Pentium II, serving demand requests on one port and prefetch requests on the other. As an optimization, we implemented the frontend as a module within the Apache server rather than as a separate process. The hint server is implemented in Java and runs on a separate machine with 932 MHz Pentium III proessor, and connects to the server over a 100 Mbps LAN. The hint server uses prediction lists generated offline using the PPM algorithm [42] over a complete 24 hour IBM server trace. The monitor runs as a separate thread of the hint server on same machine. The content munger is also written in Java and modifies the content offline (as shown in Figure 9). We have successfully tested our prefetching system with popular web browsers inluding Netscape, Internet Explorer, and Mozilla.

7.1 End to End Performance

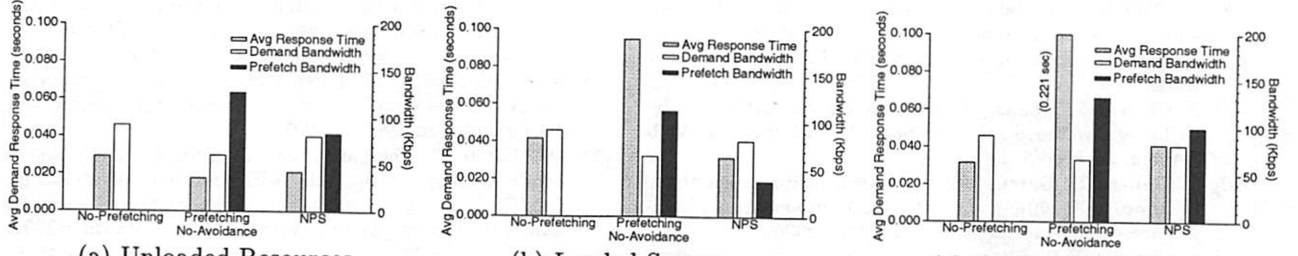
In this section, we evaluate NPS under various setups and evaluate the importance of each component in our system. In all setups, we consider three cases: (1) No-Prefetching, (2) No-Avoidance scheme with fixed *pfrate*, and (3) NPS (with Monitor and TCP-Nice). In these experiments, the client connects to the server over a wide area network through a commercial cable modem link. On an unloaded network, the round trip time from the client to the server is about 10 ms and the bandwidth is about 1 Mbps.

We use httperf to replay a subset of the IBM server trace. The trace is one hour long and consists of demand accesses made by 42 clients. This workload contains a total of 14044 file accesses of which 7069 are unique; the demand network bandwidth is about 92 Kbps. We modify httperf to simulate the execution of JavaScript as shown in Figures 9, 10 and 11. Also, we modify httperf to implement a large cache per client that never evicts a file that is fetched or prefetched during a run of an experiment. In No-Avoidance case, we set the pfrate to 70, i.e. it gets a list of 70 files to prefetch, fetches them and stops. This pfrate is such that neither the server nor the network becomes a bottleneck even for the No-Avoidance case. For NPS, we assume that each document will consist of ten files (a document is a HTML page along with the embedded objects). Thus the hint server gives out hint lists of size 10 to the requesting clients. Note that many of the files given as hints could be cache hits at the client.

Unloaded resources In this experiment, we use the setup explained above. Figure 13(a) shows that when the resources are abundant, both No-Avoidance and NPS cases significantly reduce the average response times by prefetching. The graph also shows the bandwidth achieved by No-Avoidance and Nice.

Loaded server This experiment demonstrates the effectiveness of the monitor as an important component of NPS. To create a loaded server condition, we use a client machine connected on a LAN to the server running httperf that replays a heavier subset of the IBM trace and also prefetches like the WAN client. Figure 13(b) plots the average demand response times and the bandwidth used in the three cases. As expected, even though the server is loaded, the clients prefetch aggressively in the No-Avoidance case, thus causing the demand response times to increase by more than a factor of 2 rather than decrease. NPS, being controlled by the monitor, prefetches less data and hence avoids any damage to the demand response times. NPS in fact benefits from prefetching, as shown by the decrease in the average demand response time.

Loaded network This experiment demonstrates the effectiveness of TCP-Nice as a building block of NPS. In order to create a heavily loaded network with little spare capacity, we set up another client machine running httperf that shares the cable modem connection with the original client machine, replays the same trace, and also prefetches like the original client. Figure 13(c) plots the average demand response times, demand bandwidth, and prefetch bandwidth in all three cases. The results show that when the network is loaded, No-Avoidance causes significant interference to demand requests, thereby increasing the average demand response times by a



(a) Unloaded Resources (b) Loaded Server (c) Loaded Network

Figure 13. Effect of prefetching on demand response times.

factor of 7. Although NPS doesn't show any improvements, it contains the increase in demand response times to less than 30%, which shows the effectiveness of TCP-Nice in avoiding network interference. The damage is because TCP-Nice is primarily designed for long flows.

8 Related Work

Several studies have published promising results that suggest that prefetching (or pushing) content could significantly improve web cache hit rates by reducing compulsory and consistency misses [12, 17, 24, 25, 32, 33, 42, 52]. However, existing systems either suffer from a lack of deployability or use threshold-based magic numbers to address the problem of interference. Several existing commercial client-side prefetching agents that require new code to be deployed to clients are available [39, 27, 53]. At least one system makes use of Java applets to avoid modifying browsers [20]. It is not clear however, what, if any, techniques are used by these systems to avoid self- and cross-interference.

Duchamp [17] proposes a fixed bandwidth limit for prefetching data. Markatos [36] adopts a popularity-based approach where servers forward the N most popular documents to clients. Many of these studies [17, 29, 52] propose prefetching an object if the probability of its access before it gets modified is higher than a threshold. The primary performance metric in these studies is increase in hit rate. However, the right measures of performance are end-to-end latency when many clients are actively prefetching, and interference to other applications.

Davison et. al [16] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. Servers speculatively push documents chunked into datagrams of equal size and (modified) clients use range requests as defined in HTTP/1.1 for missing portions of the document. Servers maintain state information for prefetching clients and use coarse-grained estimates of per-client bandwidth to limit the rate at which data is pushed to the client. Their simulation experiments do not explicitly quantify interference and use lightly loaded servers in which only a small fraction of clients are prefetching. Crovella et. al [12] show that a window-based rate controlling strategy for sending prefetched data leads to less bursty traffic and smaller queue lengths.

In the context of hardware prefetching, Lin et. al [34] propose issuing prefetch requests only when bus channels are idle and giving them low replacement priorities so as to not degrade the performance of regular memory accesses and avoid cache pollution. Several algorithms for balancing prefetch and demand use of memory and storage system have been proposed [6, 8, 31, 44]. Unfortunately, applying any of these schemes in the context of Web prefetching would require modification of existing clients.

9 Conclusion

We present a prefetching mechanism that (1) systematically avoids interference and (2) is deployable without any modifications to the HTTP/1.1 protocol, existing clients, existing servers, or existing networks.

References

[1] Apache HTTP Server Project. http://httpd.apache.org.

[2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[3] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, 1999.

[4] C. Bouras and A. Konidaris. Web Components: A Concept for Improving Personalization and Reducing User Perceived Latency on the World Wide Web. In *The 2nd International Conference on Internet Computing*, 2001.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE Infocom*, 1999.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *SIGMETRICS*, 1995.

[7] B. Chandra. Web Workloads Influencing Disconnected Service Access. Master's thesis, University of Texas at Austin, May 2001.

[8] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource Management for Scalable Disconnected Access to Web Services. In *WWW10*, May 2001.

[9] X. Chen and X. Zhang. Coordinated Data Prefetching by Utilizing Reference Information at Both Proxy and Web Servers. In *PAWS 2001*.

[10] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. In *2000 ACM International Conference on Management of Data*, May 2000.

[11] J. Cleary and I. Witten. "Data compression using adaptive coding and partial string matching". *IEEE Trans. Commun.*, 1984.

[12] M. Crovella and P. Barford. The Network Effects of Prefetching. In *Proceedings of IEEE Infocom*, 1998.

[13] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *SIGMETRICS*, May 1996.

[14] M. Dahlin. Technology trends data. http://www.cs.utexas.edu/users/dahlin/techTrends /data/diskPrices/data, January 2002.

[15] B. Davison. Assertion: Prefetching with GET is Not Good. Web Caching and Content Distribution Workshop, June 2001.

[16] B. D. Davison and V. Liberatore. Pushing Politely: Improving Web Responsiveness One Packet at a Time (Extended Abstract). *Performance Evaluation Review*, 28(2):43–49, September 2000.

[17] D. Duchamp. Prefetching Hyperlinks. In *Second USITS*, October 1999.

[18] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web Prefetching between Low-Bandwidth Clients and Proxies: Potential and Performance, 1999.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. HTTP/1.1. Technical Report RFC-2616, IETF, June 1999.

[20] Fireclick. Netflame. http://www.fireclick.com.

[21] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *Proceedings of the 16th International Conference on Data Engineering*, pages 3–12, 2000.

[22] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *USITS97*, Dec 1997.

[23] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable Distributed Data Structures for Internet Service Construction. In *OSDI*, 2002.

[24] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[25] J. S. Gwertzman and M. Seltzer. The Case for Geographical Push-Caching. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1995.

[26] IBM. Websphere. http://www.ibm.com/websphere.

[27] IMSI Net Accelerator. http://nct.digitalriver.com/fulfill/0002.3.

[28] Intel. N-tier Architecture improves scalability and ease of integration. http://www.intel.com/eBusiness/pdf /busstrat/industry/wp012302.pdf.

[29] Q. Jacobson and P. Cao. Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies. In *Third International WWW Caching Workshop*, 1998.

[30] V. Jacobson. "Congestion avoidance and control". In *Proceedings of the ACM SIGCOMM '88 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1988.

[31] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *OSDI*, pages 19–34, 1996.

[32] M. Korupolu and M. Dahlin. Coordinated Placement and Replacement for Large-Scale Distributed Caches. In *Workshop On Internet Applications*, June 1999.

[33] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *USITS*, 1997.

[34] W.-F. Lin, S. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. In *IEEE Transactions on Computers special issue on computer systems*, volume Vol.50 NO.11, November 2001.

[35] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *OSDI 2000*.

[36] E. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *INET 1998*.

[37] J. C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, California, 1995.

[38] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[39] Naviscope. http://www.naviscope.com.

[40] Netscape Communications Corporation. JavaScript Security. http://developer.netscape.com/docs/manuals/ communicator/jsguide4/sec.htm.

[41] A. Odlyzko. Internet Growth: Myth and Reality, Use and Abuse. *Journal of Computer Resource Management*, pages 23–27, 2001.

[42] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World-Wide Web Latency. In *Proceedings of the SIGCOMM*, 1996.

[43] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *USENIX Annual Technical Conference*, 1999.

[44] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP*, 1995.

[45] R. Rajamony and M. Elnozahy. Measuring Client-Perceived Resonse Times on the WWW. In *USITS*, 2001.

[46] Resonate Inc. http://www.resonate.com.

[47] RFC 2475. An Arhitecture for Differentiated services. Technical Report RFC-2475, IETF, June 1999.

[48] N. T. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. N. Bershad. Receiver Based Management of Low Bandwidth Access Links. In *INFOCOM*, pages 245–254, 2000.

[49] C. S. Systems. http://www.cheetah.com.

[50] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth Constrained Placement in a WAN. In *Symposium on the Principles of Distributed Computing*, Aug 2001.

[51] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A Mechanism for Background Transfers. In *OSDI*, December 2002.

[52] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. The Potential Costs and Benefits of Long Term Prefetching for Content Distribution. In *Sixth Web Caching and Content Distribution Workshop*, June 2001.

[53] Wcol. http://shika.aist-nara.ac.jp/products/wcol/wcol.html.

[54] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[55] Zeus Technology. http://www.zeus.com.

[56] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, AT&T Center for Internet Research at ICSI, http://www.aciri.org/, May 2000.

Anypoint: Extensible Transport Switching on the Edge

Kenneth G. Yocum, Darrell C. Anderson,[*] Jeffrey S. Chase, and Amin M. Vahdat
Department of Computer Science[†]
Duke University
{grant,anderson,chase,vahdat}@cs.duke.edu

Abstract

Anypoint is a new model for one-to-many communication with ensemble sites—aggregations of end nodes that appear to the external Internet as a unified site. Policies for routing Anypoint traffic are defined by application-layer plugins residing in extensible routers at the ensemble edge. Anypoint's switching functions operate at the transport layer at the granularity of transport frames. Communication over an Anypoint connection preserves end-to-end transport rate control, partial ordering, and reliable delivery. Experimental results from a host-based Anypoint prototype and an NFS storage router application show that Anypoint is a powerful technique for virtualizing and extending cluster services, and is amenable to implementation in high-speed switches. The Anypoint prototype improves storage router throughput by 29% relative to a TCP proxy.

1 Introduction

This paper presents the design and implementation of Anypoint, a new architecture for transparent communication with *ensemble sites* such as cluster-based network services. Intermediary routers at the ensemble border act to mediate communication with the ensemble, presenting it as a single virtual site to the outside network. Anypoint is implemented as a new set of functions for extensible switches.

Anypoint provides *transport switching*; it is the first general indirection approach that operates at the granularity of transport *frames*. *Transport switching* enables reliable, ordered, rate-controlled communication to the ensemble through a redirecting switch. Unlike a TCP proxy, an Anypoint switch does not terminate transport connections. Instead Anypoint switching functions transform each frame to maintain transport-layer guarantees between end nodes. Anypoint is complementary to IP-layer Internet indirection architectures such as Anycast [33] and *i3* [41].

A key goal of Anypoint is to generalize "L4-L7" server switches that support load balancing and content-aware request routing for Web server clusters. Previous work (e.g., [31, 8, 9, 18]) demonstrated the importance of content-aware request routing for Web services, and the challenges of supporting it, particularly with persistent connections [28]. Web switch architectures are limited to handle each request in a separate transport connection, or to process requests on each persistent connection serially; the former increases overheads and provides no ordering guarantees, and the latter limits concurrency and imposes head-of-line blocking for delayed packets or large requests. While these restrictions still exist for HTTP, redirection architectures that depend on them cannot extend to other services including network storage protocols, which have also been shown to benefit from content-aware request routing [5].

To overcome this challenge, we designed Anypoint for advanced IP transports with partially ordered application-level framing (ALF), as proposed by Clark and Tennenhouse over a decade ago [14]. These features are present in emerging IP transports such as SCTP [40] and DCCP [27]. We show how redirecting switches can leverage framing to enable an approach that is both more powerful and more elegant than Web switches and other solutions constrained by TCP. Our premise is that IP-based services—including network storage, general RPC-based services, and next-generation Web services using SOAP/HTTP—will migrate from TCP to these new transports. Our goal is to define a redirecting switch that accommodates pluggable indirection policies for a wide range of service protocols, not limited to HTTP over TCP. This generality is in the spirit of Active Networks [44] and subsequent proposals for extensible routers [17, 29, 38].

The contributions of this paper are to: (1) show that transport frame switching at the network edge is a powerful technique to virtualize and extend Internet services, (2) define an extensible framework and mechanisms to enable this technique, (3) present experimental results demonstrating the use of Anypoint for an NFS storage router, (4) compare the behavior of the Anypoint pro-

[*]D. Anderson is currently at Google.

[†]This work is supported in part by the U.S. National Science Foundation (CCR-00-82912, EIA-9972879, and ANI-126231), Network Appliance, and Cisco Systems, and by an equipment grant from IBM.

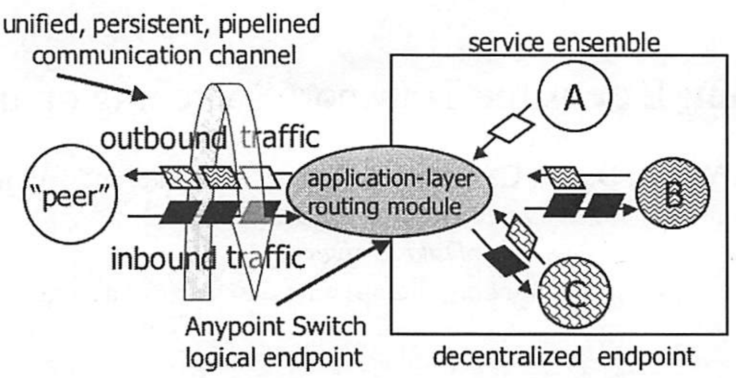

unified, persistent, pipelined communication channel
service ensemble
outbound traffic
"peer"
application-layer routing module
inbound traffic
A
B
C
Anypoint Switch logical endpoint
decentralized endpoint

Figure 1: A connection between a peer (client) and an Anypoint server ensemble. The ensemble appears to the peer as a single virtual server; an Anypoint switch at the ensemble border transforms the packets to route the connection's traffic according to policies defined by switch plugins (Application Layer Routing Modules or ALRMs). In addition to simple load balancing, an ALRM can implement content-aware routing policies to improve service performance and robustness. In contrast to current Web switches, Anypoint enables concurrent handling of pipelined requests while preserving ordering constraints at the transport layer; thus it is useful for a general class of service protocols including IP storage.

totype to an alternative structure using application-level proxies [19, 39], and (5) explore the implications for Internet service structure, extensible switches, and IP transport protocols.

2 Motivation and Overview

Figure 1 illustrates the Anypoint abstraction. An Anypoint connection allows communication between two logical endpoints: an *ensemble* of end nodes and another *peer* IP site. In a typical use, the ensemble is a server cluster and the peer is a client interacting with it through some request/response service protocol. Many connections may be active to the same ensemble. The current ensemble membership for a connection is its *active set*. Anypoint switches direct the traffic flow between each peer and its active set under the control of the ALRM plugins. This serves four related purposes:

- **Dynamic request redirection.** Anypoint switches direct inbound requests to selected servers according to service-specific policies in the ALRM. The ALRM may consider information above the transport layer, such as client identity, client location, or the nature of the request, as well as server status and traffic conditions. Content-aware policies can optimize server cache effectiveness as well as balance load [31, 18, 5], as in the NFS storage router example in Section 3.2.

- **Server resource management.** Switch-based redirection enables flexible resource provisioning in Internet server clusters. A management interface in the switch allows the service to reassign server resources with dynamic, coordinated changes to its active sets [7, 13], without relying on the client to select a new server or refresh a DNS cache.

- **Response merging.** Anypoint supports direct delivery of content to each peer from multiple ensemble nodes [22, 5]. The peer receives the response traffic on a single connection and assembles it using ordering information inserted by lightweight translation functions at the switch. The reassembly buffer resides in the peer, not in the switch.

- **Service composition.** ALRMs can act as "wrappers" to compose or extend services. For example, an ALRM might support mirroring for an ordered multicast of request traffic across replicated servers. Since the ALRM understands the service protocol, it might distribute read traffic evenly and mirror only those requests that modify service state.

This paper focuses on the first three goals, i.e., the role of Anypoint as a basis for virtualized, manageable, scalable Internet services.

Redirecting switches are controversial because the Internet architecture implicitly assumes that each IP datagram is addressed and delivered to a uniquely defined end host, running a single instance of its operating system (cf. RFC 1122 [12]). Anypoint provides a rich set of capabilities enabling an ensemble operating system and its applications to manage the ensemble as a coordinated virtual "host"—effectively a multicomputer with internal policies for handling network traffic addressed to it. Crucially, the indirection hides the ensemble's internal structure from the connection peer: the peer addresses inbound traffic to a *virtual IP address* (VIP) for the ensemble and receives outbound traffic with that VIP as the source address. This use of network address translation (NAT)—which may be damaging in other contexts—is done with the guidance and consent of the ensemble applications, and is transparent to the peer. The switch does not obscure the identity of the peer from the ensemble members.

2.1 Anypoint and the Transport

Anypoint extends IP transports to decentralized (ensemble) endpoints in a way that supports the key transport properties of rate-controlled, ordered, reliable delivery. A core principle of Anypoint is *transport equivalence.* To end nodes, an Anypoint connection is equivalent to a point-to-point connection. The burden of maintaining these properties—reorder buffering and reassembly, duplicate suppression, acknowledgment, retransmission—are handled in the usual end-to-end fashion by the end nodes. This minimizes buffering and processing overhead in the switch.

As previously stated, Anypoint assumes use of transports with framing [14]. Our purpose is not to propose a new transport, but rather to construct an indirection architecture that generalizes to a class of service protocols and framed transports. A *frame* is a variable-sized, self-contained sequence of bytes sent as a unit over a transport connection; frames are semantically meaningful and independently processed at the application layer. The service protocol uses the transport API to map its requests and responses into transport frames. Depending on the transport, a frame could span multiple transport segments (packets), or a single segment might contain multiple frames. Frame boundaries are recognizable from the transport headers, so a receiver may process frames independently as segments arrive.

Framing and partial ordering enable network-level processing of transport flows, which is useful for network adapters with protocol offload or direct-access features (e.g., RDMA), as well as for virtualization switches and routers. Frames are the granularity of transformation, redirection, and merging in Anypoint. While a service protocol may send frames over TCP, TCP imposes a total order even when the service does not require it; this precludes redirecting frames concurrently to different nodes, which could violate the delivery order. To permit effective fine-grained (e.g., content-aware) redirection, the service protocol and transport must not constrain frame order unnecessarily. However, Anypoint allows the service and transport to specify partial frame orderings if necessary for correctness. SCTP is one example of a transport that supports a partial order on frames.

Because the switch performs transport-layer processing, we envision the Anypoint switches residing in a data center or enterprise, with ensemble members colocated within a single security and routing domain. For example, transport switches must access packet state beyond the IP headers, which may be encrypted (e.g., with IPsec) across the external network. While packets from a given Anypoint connection must pass through a single intermediary, they use the service's VIP address to maintain this invariant even in the presence of asymmetric routing.

3 Anypoint Services

The Anypoint architecture factors service implementations into (1) a simple, lightweight ALRM installed as a switch extension, and (2) a server component running on the ensemble nodes, which handles complex issues such as server coordination, group membership, and recovery.

3.1 ALRMs

Anypoint ALRMs are plug-in software modules that manage Anypoint traffic through the switch. Each ALRM implements frame redirection and transformation logic for a given service. ALRMs are installed through a privileged management interface.

An ALRM is an event-driven, asynchronous, deterministic, non-blocking module supporting the interface in Table 1. ALRMs affect only traffic for specific designated ports. At most one ALRM is bound to each connection. ALRMs may access only the frame data and private structures of bounded size. We intend that the processing required per frame is statically bounded, and that the ALRM's memory references are statically verifiable.

Inside the switch, common low-level functions examine transport headers to classify packets into flows (connections), pass each frame to the *redirect()* function of the ALRM bound to its flow, and merge the transformed frames into the destination transport stream for the selected target node. These functions manipulate transport headers to extract frames from incoming transport segments and pack them into outgoing segments. The ALRM itself may perform deep processing beyond the transport headers.

ALRMs must be deterministic so they do not apply inconsistent transforms or violate exactly-once delivery for retransmitted frames. The RTX handle passed to *redirect()* indexes a unique state field for each distinct frame, enabling the ALRM to cache arbitrary state for each transformed frame. When the receiver is known to have processed an ack for the frame, the switch calls *retire(*rtx)* so the ALRM may recycle this state.

Upcalls from Anypoint switch transport functions.	
redirect(frame,id,*len,src,*rtx)	Redirect a frame on connection id. If it is inbound to the ensemble, select an ensemble member to receive it.
retire(*rtx)	Retire rtx state associated with an inactive frame.
{add,rm}_conn(id)	Add a new connection with ID, or delete a connection with ID.
alrm_{init,uninit}(void)	Start up or shutdown this ALRM.

Table 1: Registered upcall interface for an Anypoint ALRM.

3.2 Example: NFS Storage Router

To illustrate the structure of Anypoint services, we outline use of Anypoint for an L7 storage router for the NFSv3 file system protocol. We refer to this system as "Slice-lite" or *Slite* because it is a simplified form of the Slice storage architecture [5, 4]. Slite presents an ensemble of standard NFS servers as a single, unified, virtual NFS server. As in Slice, the directory tree is partitioned into subtree buckets and spread across the servers as it grows. This partitioning uses a *mkdir switching* policy, with a parameter for subtree granularity. A soft-state map tracks the assignment of subtree buckets to nodes.

The Slite ALRM uses content-aware routing to direct NFS requests to the assigned servers. Slice has previously shown that NFS is amenable to virtualization using this technique because most NFS operations apply to a single content item—a directory, name entry, or file—evident from the request type and arguments. The ALRM extracts a key identifying the subtree bucket from each request's NFS file handle at a known offset, and uses it to index the map to identify the assigned server. This indirection allows dynamic rebalancing of the buckets across the ensemble; this technique is common to cluster-based Internet services [31, 21, 36, 23]. With Slite, as in Slice and these other services, fine-grained content-aware request routing improves locality of the server caches as well as balancing load. A switch-based implementation can deliver the best latency and bandwidth, both of which are critical for file services.

Slite differs from Slice in two key respects. First, Anypoint enables Slite to run the NFSv3 protocol over a reliable, rate-controlled transport; Slice is limited to NFS/UDP, which sends one request or response per packet and relies on the RPC layer for primitive rate control and retransmission. Second, Slite is easier to deploy and it can use standard NFS servers. Although both are NFS-compliant, Slite is not Posix-compliant because it does not support *readdirplus*, *link*, and *rename* operations that cross server boundaries. While still compatible with Anypoint, this coordination requires the more comprehensive Slice approach. We use Slite for simplicity.

The Slite ALRM redirects some complex operations to a designated back-end server derived from the *coordinator* in the Slice architecture. In particular, the coordinator executes namespace operations (*mkdir*, *rmdir*, *lookup*) on name entries that cross servers (*switched directories*). The coordinator uses a write-ahead intention logging protocol for failure atomicity [4]. To simplify the ALRM further, our prototype indirects *lookup* through the coordinator, which maintains an index of switched directories. These redirect cases are easy to encode in the ALRM.

s.una	CSN of the oldest unacknowledged frame.
s.last	CSN of the last frame.
s.next	LSN of the next frame.
s.hole	CSN of the oldest pending sequence hole from a source.
s.lastgap	CSN of the newest hole at the time of the last send to a sink.

Table 2: **Endpoint table entry**. For an outbound flow, s is a source, and the fields pertain to frames transmitted by s. For an inbound flow, s is a sink, and the fields pertain to frames received by s.

4 Inside the Anypoint Switch

This section outlines the transport switching functions within an Anypoint switch. Without loss of generality we consider the traffic on a single connection. This traffic consists of two partially ordered flows of frames passing through the switch: one flow *inbound* to the ensemble and one flow *outbound* from the ensemble. The n ensemble members in the connection's active set are *sources* for the outbound flow and *sinks* for the inbound flow.

We make the following assumptions about the transport. The transport senders on the end nodes mark the transmitted frames for each flow with *frame sequence numbers* (FSNs) that are unique within the flow, monotonically increasing, and consecutive. FSNs are the basis for reliable at-most-once delivery. Frames arriving at a receiver are delivered in FSN order, and the receiver uses FSNs to generate cumulative acks in the return flow. The transport limits the number of outstanding unacknowledged frames to a frame flow window w.

frame.csn	CSN for this frame.
frame.lsn	LSN for this frame.
frame.source	Source node ID (outbound).
frame.sink	Sink node ID (inbound).
frame.ack	Has receiver acknowledged?
frame.hole	Is this frame a pending hole?
frame.link	CSN of next chain entry.

Table 3: **Frame ring entry.**

The *transport equivalence* property means that end nodes do not distinguish Anypoint connections from point-to-point connections using the same transport. This implies that each participating node views the frames from each flow in a local FSN space, since it believes (at the transport layer) that it is the exclusive owner of its end of the connection. Since there is only one connection peer, it defines a global FSN space of *connection sequence numbers* (CSNs) for both flows in the connection. The switch's sequencing functions translate between the end nodes' FSN spaces—the *local sequence numbers* (LSNs) understood by each ensemble member and CSNs understood by the peer.

The Anypoint switch transforms the frames flowing through it to split inbound traffic across the LSN spaces of the n sinks, and to merge outbound traffic from the LSN spaces of the n sources into the peer's CSN space. Correct translation of the sequence numbers is the key to extending the transport's end-to-end ordering, duplicate suppression, acknowledgment, and retransmission functions to Anypoint connections because *it enables reassembly buffering and retransmission at the end nodes rather than in the switch.* The switch also coordinates the transport mechanisms for ordering and reliable delivery, and propagates rate control signals to ensure that each Anypoint connection behaves correctly with respect to flow control and congestion.

4.1 Per-Flow State

The Anypoint switch maintains per-flow control state proportional to the number of unacknowledged frames. This state consists of a *frame ring*—a circular queue of w frame entries—and an *endpoint table* with an entry for each active set member, as shown in Figure 2. Table 2 summarizes the state in each endpoint table entry. The frame ring contains an entry for each active frame in the sliding window ranging from the oldest active frame ($flow.left$) to the highest numbered frame ($flow.left + w$) eligible for transmission into the frame flow window. Entries become active as new frames arrive, and inactive as the left edge of the window passes them.

Every frame has a unique CSN; the ring entry for a frame

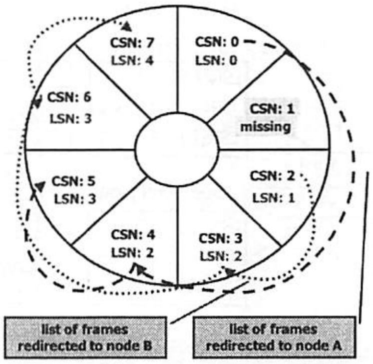


Figure 2: The Anypoint switch has one frame ring per flow, each of size w. In this example the ALRM redirects inbound frames to either sink A or sink B. The network has dropped CSN 1, so the switch inserts a gap of length 1 into each sink's LSN space. When CSN 1 arrives, the ALRM may direct the switch to deliver it to either A or B.

flow.left	CSN of the oldest frame for which the sender is not known to have received an ack.
flow.una	CSN of the oldest frame for which the receiver is not known to have sent an ack.
flow.next	CSN of this flow's next frame.
lastgap	CSN of the newest hole (inbound).
hole	CSN of the oldest hole.

Table 4: **Per-flow state variables.**

may be retrieved in constant time by indexing from the frame's CSN in the obvious fashion. Table 3 summarizes the frame ring entry, and Table 4 summarizes per-flow state variables maintained to index the frame ring. CSNs are also used to link frame entries in *frame chains* in CSN order. For outbound flows, n frame chains link the frames from each source s, including the holes originating from s. The chain for each source s is rooted in $s.una$. For inbound flows, the n frame chains link the active frames destined for each sink, with a separate chain for pending sequence holes (see below). Every active frame is linked into exactly one chain.

Given these data structures, many aspects of state maintenance for the endpoint table, frame ring, and flow variables are straightforward. The discussion below focuses on the more interesting aspects.

4.2 Sequencing and Acknowledgments

The switch translates frame sequence numbers between CSNs and LSNs as frames pass through it. This is trivial

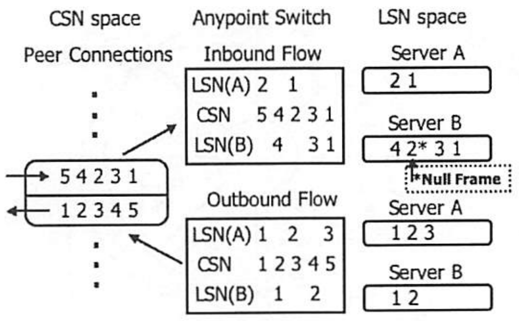


Figure 3: The Anypoint switch translates between the CSN space of the peer connection and each server's LSN space. The network reorders frames 2 and 3 on the *inbound* flow; a gap is inserted into server B's LSN space to maintain ordering if frame 2 is redirected to server B. However, frame 2 is redirected to server A, and a null frame is sent to server B.

in the common case of frames arriving in order: if the flow is outbound (merge), assign CSN $flow.next$; if it is inbound (split), assign LSN $sink.next$ for the frame's destination sink. Figure 3 shows resequencing between CSNs and LSNs for a single peer connection to two servers.

The frame ring and chains allow efficient handling of acks. For inbound flows, the sinks encode their outgoing acks as LSNs. To convert an LSN ack from $sink$ to a CSN for the peer, the switch traverses the frame chain for $sink$ from $sink.una$ and examines each frame's lsn to determine if the ack covers it. The ack passed through to the peer is the CSN of the oldest frame not acknowledged by any sink s: $min(s.una)$. For outbound flows, on receiving a CSN ack from the peer, the switch traverses the frame ring from $flow.una$ to identify frames covered by the ack and update their $source.una$. The LSN ack passed through to each source is its $source.una$.

The difficult sequencing cases involve reordered or dropped packets, which create sequence holes. To allow ordered, reliable delivery at the receiver, the switch must pass any sequence holes through to the receiver's sequence space. For example, if an **outbound** frame's LSN $frame.lsn > source.next$ for its source, then one or more frames from that source are delayed. The switch passes the gap through to the peer by assigning the frame CSN $flow.next + (frame.lsn - source.next)$, marking the intervening frame entries starting at CSN $flow.next$ as holes from that source, and updating $flow.next$ and $source.next$. On the other hand, if $frame.lsn < source.next$, then this is a delayed frame that fills an existing hole. To identify the frame's CSN, follow the source's frame chain forward from $source.hole$ to locate the hole. The switch then places the CSN in the frame before forwarding it to the the peer.

The more difficult case occurs when an **inbound** frame is delayed. Since the switch does not know which sink will receive a given delayed inbound frame, it must reserve an LSN in the local space for any $sink$ that receives a frame numbered after a pending hole. For each inbound frame, assign the LSN as $sink.next + h$, where h is the number of pending holes created since the last frame sent to the destination $sink$. The common case $h = 0$ is easily recognized with a check that $sink.lastgap = flow.lastgap$. Else h is the number of pending hole frames f with $f.csn > sink.last$; these are found by traversing the frame chain rooted at $flow.hole$. Note that each pending hole is visited exactly once for each destination sink that receives a frame while the hole is pending; inbound holes create at worst $O(n)$ extra work within the switch.

When an inbound frame destined for $sink$ arrives to fill a hole at CSN i, traverse the frame ring forward from i, visiting each entry. For each sink s, let f be the entry for the first non-hole frame encountered that was destined for s (if any frames were sent to s since the sequence gap opened at i). If $s = sink$, then redirect frame i to s. Else, send a null frame (*redirect patch*) to s to fill the gap in its local sequence space (see Figure 3 for an example). In each case, the LSN for the sent frame is $f.lsn - h$, where h is the number of holes encountered before f in the traversal, including the hole at i. If no frame for $sink$ is encountered before the end of the traversal (CSN $flow.next$), then redirect i to $sink$ with LSN $sink.next$. Again, the number of redirect patches per hole grows with the number of frames arriving while the hole is pending, and is bounded by $n - 1$ and w.

4.3 Rate Control

This section examines the role of the Anypoint switch in coordinating rate control. This discussion considers the traffic from the switch's viewpoint: because the switch must control each endpoint's rate independently, it views the traffic to or from each endpoint (the n active set members plus the peer) as distinct flows. (This contrasts with the previous section, in which the peer's view of a connection is a pair of flows, one inbound to the ensemble and one outbound from the ensemble.) The switch merges a *fan in* of n flows—one from each ensemble member—into the connection's outbound flow to the peer, and splits the peer's inbound flow into a *fan out* of n flows. As flows split and merge, the switch propagates rate control signals to avoid overflowing any receiver or network path. Transport equivalence implies that end nodes do not change their rate control policies to use Anypoint; the end nodes are not aware that a split or merge is occurring. Instead, it is the switch's responsibility to transform and coordinate these signals to induce the correct local behavior from the sources and produce the desired global outcome.

The switch observes rate control signals flowing through it, and can determine if forwarding a frame violates rate limits to the receiver. It can also send rate control signals to any sender. *Flow control* signals proactively limit the rate of the source; we assume that the transport allows a receiver to rate-limit a source by advertising a flow window. A switch may manipulate these windows to suit its needs [25]. *Congestion* signals cause a sender to reactively reduce its rate. For example, if the switch drops a packet, a TCP-friendly sender interprets the event as congestion in the usual fashion.

The policy choice is to determine how the switch uses these rate control signals to respond to observed conditions. But the switch cannot predict how the ALRM will route inbound traffic to the ensemble sinks, or what portion of the bandwidth back to the peer will be needed for each source. In either direction, it may optimistically oversubscribe the windows, conservatively rate-limit senders to avoid any overflow, or select any point on the continuum between these extremes. For example, for inbound traffic it may optimistically advertise the sum of the active set flow windows to the peer, or conservatively advertise the minimum window from any sink. For outbound traffic, it may advertise the peer's full window to each ensemble source, partition it evenly among the sources, or overcommit it to an arbitrary degree.

The conservative approaches may limit connection throughput, while the optimistic approaches may cause the switch to overflow a receiver or network path, forcing it to drop packets. A dropped inbound packet induces the peer to throttle its sending rate to the entire ensemble, even if just one sink overflows. The peer's inability to distinguish among ensemble nodes is fundamental to the Anypoint model; we accept it because we assume that the network and memory within the ensemble are well-provisioned in the common case, and aggregate throughput is more important than bandwidth from the peers when the ensemble is overcommitted. For outbound traffic, congestion on the path to the peer results in a lazy throttling of individual sources in the usual fasion.

4.4 Discussion

The switch mechanisms described in this section illustrate several key points about the Anypoint architecture. Most importantly, *transport equivalence* says that Anypoint does not affect the transport connection semantics perceived by the end nodes. This architectural choice yields several benefits:

- End nodes use the same transport code for point-to-point and Anypoint connections, and do not distinguish between them. All Anypoint-specific functions are local to the switch.

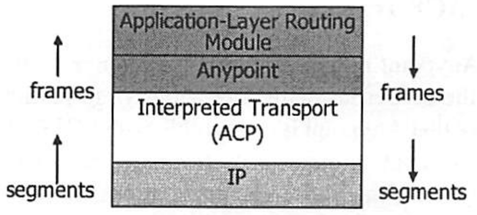


Figure 4: The switch extracts frames from incoming network segments and packs them into outgoing segments.

- Transport equivalence helps to ensure that Anypoint is compatible with all clients and servers implementing the transport, conforms to congestion conventions, and composes well with other components of the Internet architecture. For example, given appropriate ALRM support, both ends of an Anypoint connection could be ensembles, or Anypoint ensembles could nest arbitrarily in a hierarchy.

- Transport functions for sequencing and reliable at-most-once delivery continue to operate in an end-to-end fashion. The Anypoint switch never buffers data for rate control or transport reassembly; its role is to transform frames to coordinate these functions at the end nodes. The switch buffers packet data only as needed for port queues and frame assembly. This improves scalability of the switch architecture.

Although the switch maintains per-flow control state, it is bounded by the flow window w. Because acknowledgments and buffering remain end-to-end in Anypoint, a failed switch does not lose user data. However connections maintained by the failed switch must be re-initiated for the service to recover a failed session. Support for session recovery is common in new service protocols including iSCSI and DAFS [16].

Note also that no mutable state is shared across connections within an Anypoint switch. Thus an Anypoint switch design could spread frame processing load across processors at each external switch port. An ensemble may also partition communication traffic from different peers across multiple switches to further improve scalability.

5 Anypoint Prototype

We prototyped a host-based Anypoint switch as a set of kernel extensions to FreeBSD, implementing the transport switching mechanisms and ALRM interface. The prototype consists of 2080 lines of C code. We also implemented Anypoint ALRMs for the Slite service described in Section 3.2 and a simple clustered counter service for microbenchmarking.

5.1 ACP Transport

The Anypoint model can apply to a range of transports with the properties defined previously. In particular, we believe that Anypoint is compatible with SCTP. However, SCTP is a new protocol with complex features unrelated to Anypoint, and SCTP implementations are not yet fully mature. To experiment with Anypoint, we implemented a simple framed transport with a few hundred lines of code by reusing the FreeBSD TCP implementation, whose behavior is stable and reasonably well understood. We refer to this as Anypoint Control Protocol (ACP), although its functions are not Anypoint-specific.

ACP adds a shim with framing support based on a subset of the expired upper-layer framing (TUF) proposal [10]. New code at the kernel socket layer supports ACP sockets using a UDP-like message interface (*sendto, recvfrom*). Each message is sent as a frame, and frame boundaries are preserved at the receiver; this made it easy to run NFS over ACP for the Slite experiments.

An ACP segment is identical to a TCP segment, except that ACP adds one or more framing headers to each segment's data to partition it into an integral number of frames with consecutive FSNs as defined in Section 4. Like TCP, ACP preserves the send order for all of a connection's frames routed to a given end node. However, ACP does not specify an order among frames to or from different ensemble nodes. ACP differs from TCP primarily in that ACP is not defined to require this ordering. ACP's mechanisms for reliable delivery and rate control are indistinguishable from TCP (4.4 BSD Reno).

ACP segments are *self-describing*. The first frame header in each segment is aligned with the segment header so that an Anypoint switch can recognize frames even when segments arrive out of order. Each frame header contains a length field giving the offset of the next frame in the segment, if any. Our testbed network uses 9000-byte "Jumbo" Ethernet packets, and the maximum ACP frame size is 8936 bytes, which is less than the Maximum Segment Size (MSS). The ACP prototype never splits application frames across segments, although it may combine multiple frames in a single segment when space allows. In practice, this means that ACP often sends short segments, but it also frees the prototype from the need to "chunk" or reassemble frames across segment boundaries. Our ACP prototype does not support path MTU changes.

5.2 Switch Prototype

Figure 4 shows the protocol stack at the switch. Arriving IP packets are interpreted by a *veneer* layer that recognizes the transport, classifies flows, and extracts frames. The Anypoint layer maintains connection-related state, active set membership, and ALRM bindings as described in the previous sections. The TCP-derived ACP receiver uses TCP byte sequence numbers for reordering, and ignores FSNs. The switch also translates byte sequence numbers and caches them in the frame ring. It uses this state to identify the frames covered by acks, which are encoded as byte sequence numbers rather than FSNs.

The switch prototype validates TCP checksums for incoming ACP segments and recomputes a fresh checksum from scratch for transformed outgoing segments, using network cards with TCP checksum offloading. Redirect patches are carried in each ACP segment as a monotonically increasing sequence number. This field, updated by the switch for inbound flows, indicates that all data with sequence numbers less than this number may be delivered to the application. The switch has limited support for dynamic changes to the active set; it can remove failed servers, but it cannot add servers.

6 Experimental Results

This section presents results from our host-based Anypoint/ACP prototype and the Slite/NFS server cluster. The experiments explore the overhead and bandwidth the host-based Anypoint switch, frame processing costs, memory requirements, interactions with TCP rate control, and scaling and response time of Slite/NFS.

We also compare behavior of the Anypoint switch with an alternative service structure using a *redirecting proxy* that terminates incoming client connections and relay traffic over connections maintained between the proxy and the servers. Our proxies are implemented at the application level for TCP or UDP. The TCP proxy uses a blocking *select* to relay data between the peer and ensemble.

The Anypoint testbed consists of Dell PowerEdge 4400s with 733 MHz Pentium-III CPUs and 256 MB RAM, running FreeBSD 4.4. Each node has an Alteon Gigabit Ethernet NIC with hardware checkum offloading, connected to an Extreme Summit 7i switch. Unless stated otherwise, our microbenchmark tests use 4KB transport frames, 128 KB socket buffers, 9KB (Jumbo) segment/MTU sizes, and delayed acks. The Anypoint switch uses a frame window w of 384 entries. Each Slite NFS server is fitted with eight 18 GB 10,000 RPM Seagate Cheetah drives over two dual-channel Ultra-160 SCSI controllers.

6.1 Memory and CPU Overheads

First we explore the raw performance of the Anypoint switch and compare its CPU and memory utilization to a TCP proxy. The microbenchmark suite consists of simple user-level programs that open TCP or ACP sockets.

Figure 5 shows the peak aggregate throughput for 8 inbound streams (outbound results are identical) for the

Anypoint switch and TCP proxy. The active set for each stream is a single ensemble server; no merging or splitting occurs. We vary the frame size from 8KB to 1KB (2KB decrements), increasing the number of frames per segment on the x-axis. With 8KB frames, the TCP proxy's aggregate bandwidth is CPU-limited at 63MB/s, while the Anypoint switch is NIC-limited at 106MB/s. To factor out copying costs in the user-level proxy, the *Anypoint copy* lines show the performance of the Anypoint switch with two copies added to the critical path. Even with the copying, avoiding full termination in Anypoint yields an average 29% improvement in peak bandwidth. The declining bandwidth with increasing frames per segment quantifies the effect of per-frame processing costs.

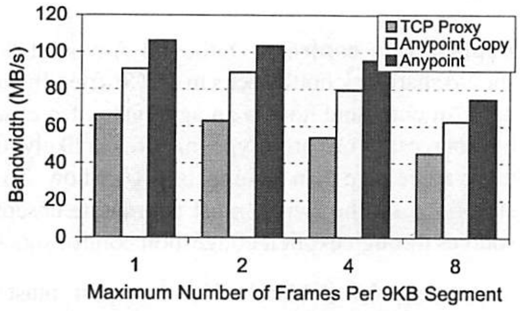


Figure 5: Aggregate bandwidth through the Anypoint switch and TCP proxy with increasing number of frames per segment.

Figure 6 shows the memory overhead of the TCP proxy and Anypoint switch for 8 simultaneous inbound single-server connections as the round trip time increases between client and ensemble (using dummynet [35]). Anypoint memory usage is determined by the flow window w and is independent of the number of servers active per connection. In contrast, a TCP proxy's memory usage scales with the aggregate bandwidth-delay product (BDP) for outbound flows. For inbound flows, proxy memory usage is inversely proportional to the BDP because an increasing share of the flow window is in transit in the network rather than buffered at the proxy.

Next we investigate splitting and merging of a single connection. Figure 7 shows throughput for a single connection that is either outbound (merged) or inbound (split) from/to four servers. The ALRM round robins the inbound 4KB frame stream across the ensemble. Throughput for inbound and outbound flows is nearly identical. As the round-trip time (RTT) and BDP increase, the outbound bandwidth to the peer is limited by the peer's receive window for both Anypoint and the TCP proxy. The Anypoint switch conservatively splits the peer's flow window evenly among the servers, each of which achieves the same share of the outbound bandwidth. For the inbound case, the TCP proxy is limited by its own receive window as BDP increases. In contrast, Anypoint's inbound bandwidth is limited by its conservative flow window advertisement, which is the the minimum of the server's advertised windows.

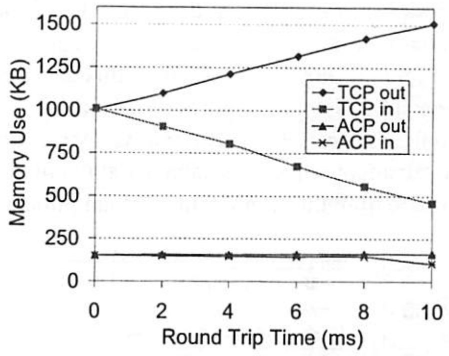


Figure 6: Total memory usage of a TCP proxy versus an Anypoint switch as a function of round-trip time. There are 8 simultaneous connections.

The Anypoint inbound and outbound flows achieve lower throughput than the TCP proxy at 2ms RTT. This effect is evident even with one server per connection. It occurs because the TCP proxy acknowledges data immediately, even before forwarding it to the receiver. This trades end-to-end reliable delivery and proxy buffer memory for improved bandwidth in this case. This effect diminishes with increasing RTT.

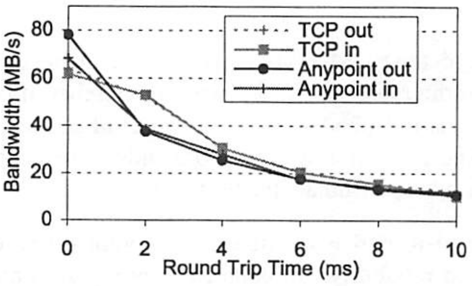


Figure 7: Bandwidth for single inbound or outbound connections with an ensemble of four servers.

6.2 Layer-4 Informed ALRMs

We now explore integrating layer-4 information into ALRMs with *speed-sensitive steering*, which always selects servers with open flow windows for inbound frames instead of using strict round robin, balancing inbound traffic across the ensemble more effectively. This allows the Anypoint switch to optimistically advertise the *sum* of the ensemble's flow windows to the peer.

We now compare the inbound throughput of a speed-sensitive steering ALRM versus the TCP proxy. In this experiment there are four servers, server socket sizes (flow windows) are 32KB, and the MTU is 1500 bytes. One server incurs a variable delay between processing frames. Figure 8 shows throughput as this delay increases to 0.75ms. The TCP proxy gates the receive rate of every server to the slowest server. But the Anypoint connection can take advantage of excess capacity at the other servers and maintain high throughput during load imbalances.

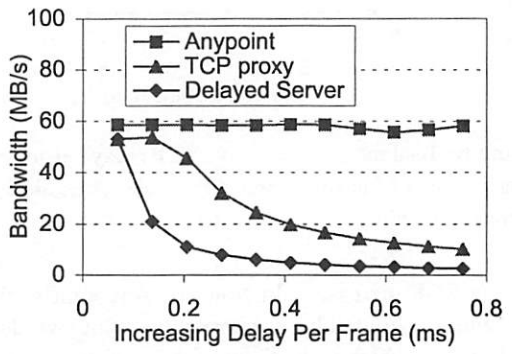


Figure 8: With speed-sensitive steering, the ALRM redirects inbound frames to servers with sufficient capacity to handle them. Here one server incurs an increasing CPU delay to process a frame. The TCP proxy gates the receive rate of each server to that of the bottlenecked server.

6.3 Observations

Anypoint offers three advantages relative to the TCP proxy:

- **Efficiency.** Memory use is bounded independent of traffic rates, and scales with the number of connections independent of the number of active servers. The Anypoint switch also avoids processing overheads to terminate the protocol.
- **End-to-end guarantees.** Anypoint offers end-to-end reliability. In contrast, a proxy acks data that it has not yet delivered to the receiving end node. Also, note that the proxy's delivery order is the same as Anypoint's, which is not the ordering specified by its transport (TCP).
- **Layer integration.** Anypoint allows a continuum of redirection policies that consider Layer 4 state, e.g., for speed-sensitive steering during periods of unbalanced load.

TCP splicing is one technique to reduce the runtime overheads for a proxy [19], and is amenable to switch-based implementations. This technique is related to Anypoint's sequence number translations to short-circuit protocol processing. However, the Anypoint transport model is fundamentally different.

Interestingly, inbound Anypoint flows in our prototype may slow down relative to a TCP proxy as the ensemble size grows, due to an interaction between the transport's congestion control and acknowledgments from the ensemble. The Anypoint switch merges acks from the ensemble nodes and sends cumulative acks to the peer. If the servers return acks out of order, the switch must delay them to avoid inciting a fast-recovery reaction on the peer, causing it to reduce the congestion window (TCP Reno and later presume that duplicate acknowledgments indicate lost data). Delaying these acks can negatively impact the acknowledgment clocking, lowering throughput.

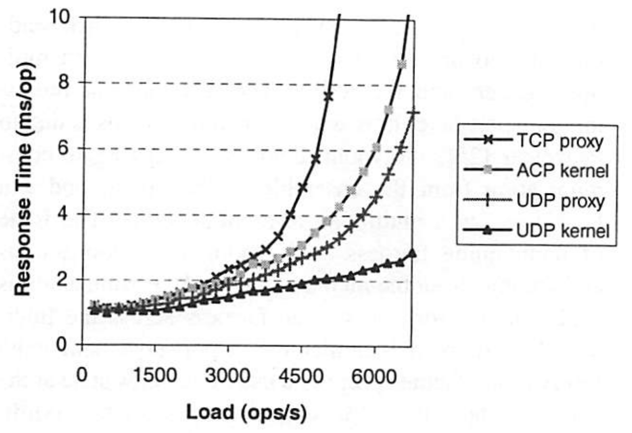
After these experiments we can make a number of observations about desirable features for Anypoint-compatible transports:

- **Explicit rate control.** Outbound Anypoint flows should share link bottlenecks in a TCP-friendly manner. An outbound flow is an aggregate of n ensemble sources; in our prototype this flow is likely to be more aggressive than a competing TCP flow. To ensure fairness, the switch must coordinate ensemble sources through explicit congestion control signals.
- **Selective acks (SACK).** The transport must divorce congestion behavior from reliability. Triple-duplicate cumulative acks are common and meaningless as congestion indicators for Anypoint communication.
- **Flexible flow control.** The switch can optimistically or conservatively manage the flow windows as described in Section 4.3. If the switch conservatively distributes the peer's advertised receive window across the ensemble sources, it should be able to revoke unused window allocations and redistribute them to active sources. Alternatively, ensemble members could bid for the peer's receive window by advertising to the switch the amount of data they wish to send.

6.4 NFS Storage Router

In this section we evaluate the performance of the Slite NFS storage router. We compare an Anypoint Slite storage router (described in detail in Section 3.2) to user and kernel-level proxy alternatives.

The kernel-level (fastpath) configurations employ an Anypoint ALRM module to redirect frames. The ALRM sends most frames directly to servers; some require additional processing and go to the *coordinator*, which is just another server to the Anypoint layer. This category uses either an ACP or UDP transport. The UDP fastpath uses a simple kernel hook to intercept UDP packets and advertise them via the Anypoint interface to the Slite ALRM.



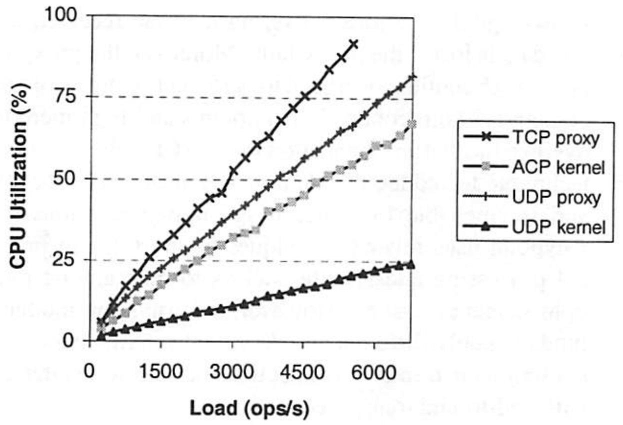


Figure 9: Slite latency and switch CPU utilizations as a function of offered load for varying intermediary configurations.

We use one coordinator proxy and four back-end NFS servers, exporting independent drives for a total of 32 NFS volumes (unified into a single logical volume). All machines run FreeBSD 4.4. Delayed acks are disabled to get good performance from the NFS RPC protocol over TCP, and the system is configured to use 8KB transport frames.

Figure 9 plots results from these four configurations, using the Fstress NFS benchmark [3] configured to generate a Web server's file system load. We measure average operation response time and CPU utilization at the switch with increasing request rate, plotted on the left and right, respectively. The TCP proxy has the lowest peak throughput and becomes CPU saturated at 5500 ops/s. The Anypoint switch shows a 29% improvement in throughput relative to the TCP proxy.

Note that the user-level UDP proxy gives better response times than the Anypoint switch, even though the Anypoint CPU overhead (shown in the right graph of Figure 9) is lower at any given request rate. We believe Anypoint/ACP is limited by the interaction between cumulative acks and congestion control described in the previous section.

7 Related Work

There is a large body of work on incorporating architecturally correct support for indirection into the Internet. The design alternatives span layers of the Internet architecture and all levels of its implementation. Indirection may apply to names at any layer (e.g., URNs or URLs, domain names, IP addresses, or abstract names [43, 1]). A given request may route through multiple intermediate nodes, as in hierarchies, meshes, or overlays for Web content caching [46]. Request routing policies may reside in application-level intermediaries (e.g., Web proxy caches), DNS name servers (e.g., Akamai and other CDNs), in the clients [45] or servers [8, 9] themselves, or in the network switches or routers.

IP-layer indirection. Anycast [33] and *i3* [41] support best-effort packet delivery with indirection at the IP layer. Anypoint operates at the transport layer and above for a related but different purpose. While Anycast is useful for binding to service sites across the wide area, Anypoint enables stateful, reliable, and congestion-aware connections to an ensemble at a logical site.

Web switches and persistent connections. Anypoint is similar in goals and concept to L4-L7 Web server switches, which implement server load balancing (SLB) and/or content-based routing (CBR) for Web server ensembles. The benefits of Web server traffic management are well-established from research studies (e.g., [31]) and commercial experience. Anypoint offers a more powerful mechanism for handling *persistent connnections*, as in HTTP 1.1, in which a single transport connection carries a stream of requests to simplify congestion control and amortize connection setup costs [28]. Web switches supporting HTTP 1.1 may route all requests on a given connection to a single server, or else they use *connection handoff* [8, 32, 37] to migrate the connection to a different server between requests. The first approach does not support CBR and the second forces the server ensemble to process and respond to the requests in strict sequence, as mandated by HTTP 1.1. Anypoint enables independent, pipelined processing of the requests, to generalize redirection to a broader class of service protocols that do not impose this constraint. This is important for fine-grained requests, e.g., storage protocols such as iSCSI or NFS, or RPC protocols such as RMI.

Redirecting proxies. Section 6 compares Anypoint to redirecting *proxies* that terminate incoming client connections and relay traffic over connections maintained between the proxy and the servers. Proxies do not preserve end-to-end transport semantics because the proxy may ac-knowledge data before delivering it to the receiver, and this data is lost if the proxy fails. Moreover, the proxy incurs a high runtime overhead to perform full protocol processing for both connection endpoints and high memory overhead to buffer connection data. TCP splicing is one technique to reduce the runtime overheads [19, 15, 39], and is amenable to switch-based implementations [6]. Anypoint uses related techniques to short-circuit protocol processing and rewrite packets to map among multiple sequence spaces. However, the Anypoint model is fundamentally different: an Anypoint intermediary does not terminate transport connections or otherwise interfere with end-to-end transport functions.

Active Networks and extensible routers. Anypoint is related to Active Networks [44], in which dynamic packets carry behavior (*capsules*) that execute on routers. Like Active Networks, the Anypoint switch architecture defines a powerful new capability for extending switches and routers through simple, clean abstractions, meeting needs that are currently served in an ad hoc fashion. However, in contrast to capsules, Anypoint extensions—ALRMs—are installed by an authorized configuration tool, are relatively constrained in their actions, and have known, bounded resource and interface requirements. Thus they are more closely related to *switchlets* in the SwitchWare architecture [2]; Anypoint enables such extensions to perform application-layer functions while preserving transport semantics. Another extensible router architecture is Click [29]; Anypoint could run as a new set of transport-layer and application-layer modules within the Click framework. Anypoint does not require or benefit from the richer interfaces of the NodeOS [34] framework for extensible routers, which supports multiple execution environments similar to those offered in a general-purpose operating system.

Group communication and multicast. Anypoint is similar to group communication in that it supports communication with a dynamic ensemble addressed in a unified way. However, the basic Anypoint abstraction is less powerful in that it supports an indirect frame unicast rather than ordered multicast. The Anypoint framework is powerful enough to allow an ALRM to implement ordered multicast using an approach similar to Amoeba [24], in which a single intermediary for each connection acts as a sequencer.

Scalable Reliable Multicast(SRM) [20] is similar to an Anypoint intermediary in that it manipulates sequence numbers and is based on application-level framing. However, SRM and Anypoint have different goals: Anypoint enables application-layer routing while SRM does not, and Anypoint is not limited to a multicast model.

Traffic shaping and congestion management. One function of an Anypoint intermediary is to control sending rates to preserve the traffic balance between multiple senders and receivers. The technique of rewriting transport-layer flow window advertisements is due to Packeteer [25]. Anypoint connections aggregate communication from the ensemble to the client, and can be viewed as a multipoint-to-point session. The issue of maintaining fairness both among ACP connections and among ensemble members on an ACP connection is analagous to work on session fairness across the Internet [26]. Anypoint is similar to transport-layer bandwidth reservation schemes [42] in its use of per-flow state at the network edge. It is also related to schemes for coordinating congestion control across multiple flows with the same source or destination, when those schemes are applied in intermediaries [11, 30].

Multihoming. SCTP includes support for *multihoming*, which is similar to Anypoint in that a single connection may deliver traffic to a site through multiple IP endpoints. Anypoint differs from multihoming in that it does not require these multiple IP endpoints to coordinate through shared memory, and it uses multiple IP endpoints concurrently for the same connection.

8 Conclusion

Anypoint is the first architecture to enable switching at the granularity of transport frames in extensible routers at the edge of the network. This approach allows service-specific application plugins (ALRMs), residing in the router, to coordinate request/response flows to and from the multiple nodes in an ensemble. These plugins may support dynamic request redirection, response merging from multiple servers, and other extensions for network services based on a partially ordered, framed IP transport.

Anypoint provides transport-layer guarantees including partial ordering, rate control, and reliable at-most-once delivery without the overhead to terminate the transport protocol in the switch. Experimental results with a host-based Anypoint prototype show that Anypoint is a powerful mechanism for traffic management and virtualization in server clusters. We present results from an Anypoint switch under various network conditions, showing that buffering overheads in the Anypoint intermediary are significantly lower than an application-level proxy. Results from an Anypoint-based NFS storage router show that Anypoint supports scalable services transparently to clients.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Peter Honeyman, for helpful critiques and suggestions.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.

[2] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, May/June 1998.

[3] D. Anderson and J. Chase. Fstress: A flexible network file service benchmark. Technical Report CS-2002-01, Duke University Department of Computer Science, January 2002.

[4] D. C. Anderson and J. S. Chase. Failure-atomic file access in an interposed network storage system. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2002. To Appear.

[5] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. *ACM Transactions on Computer Systems (TOCS) special issue: selected papers from the Fourth Symposium on Operating System Design and Implementation (OSDI), October 2000*, December 2001.

[6] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proceedings of IEEE Infocom 2000*, March 2000.

[7] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[8] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based Web servers. In *Proceedings of USENIX'99 Technical Conference*, 1999.

[9] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of USENIX Technical Conference*, June 2000.

[10] S. Bailey, J. Chase, J. Pinkerton, A. Romanow, C. Sapuntzakis, J. Wendt, and J. Williams. Internet Engineering Task Force, Internet draft: TCP ULP Framing Protocol (TUF), November 2001.

[11] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proceedings of the ACM Conference on Communications Architectures and Data Communication (SIGCOMM)*, September 1999.

[12] B. Braden. Internet Engineering Task Force, Network Working Group, RFC 1122: Requirements for Internet hosts – communication layers, 1989.

[13] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[14] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, 1990.

[15] A. Cohen, S. Rangarajan, and H. Slye. The performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS)*, October 1999.

[16] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

[17] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins – a modular and extensible software framework for modern high performance integrated services routers. In *Proceedings of the ACM Conference on Communications Architectures and Data Communication (SIGCOMM)*, September 1998.

[18] R. P. Doyle, J. S. Chase, S. Gadde, and A. M. Vahdat. The trickle-down effect: Web caching and server request distribution. *Computer Communications: Selected Papers from the Sixth International Workshop on Web Caching and Content Delivery (WCW)*, 25(4):345–356, March 2002.

[19] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *Proceedings of USENIX Technical Conference*, pages 327–334, January 1993.

[20] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[21] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[22] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 97)*, June 1997.

[23] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, pages 319–332, October 2000.

[24] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system.

In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, pages 220–230, May 1991.

[25] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer. TCP rate control. *ACM SIGCOMM Computer Communication Review*, 30(1), January 2000.

[26] P. Karbhari, E. W. Zegura, and M. H. Ammar. Multipoint-to-point session fairness in the Internet. In *Proceedings of IEEE Infocom*, 2003.

[27] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Internet Engineering Task Force, Internet draft: Datagram Congestion Control Protocol(DCCP), November 2001.

[28] J. Mogul. The case for persistent HTTP connections. In *Proceedings of the ACM Conference on Communications Architectures and Data Communication (SIGCOMM)*, pages 299–313, September 1995.

[29] R. Morris, E. Kohler, J. Jannotti, and M. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, December 1999.

[30] D. Ott and K. Mayer-Patel. A mechanism for TCP-friendly transport-level protocol coordination. In *Proceedings of USENIX Technologies Conference*, June 2002.

[31] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenopoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[32] A. E. Papathanasiou and E. V. Hensbergen. KNITS: Switch-based connection handoff. In *Proceedings of IEEE Infocom*, June 2002.

[33] C. Partridge, T. Mendez, and W. Milliken. Internet Engineering Task Force, RFC 1546: Host anycasting service, November 1993.

[34] L. Peterson, Y. Gottlieb, S. Schwab, S. Rho, M. Hibler, P. Tullmann, J. Lepreau, and J. Hartman. An OS interface for active routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[35] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, January 1997.

[36] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, availability and performance in Porcupine: a highly scalable cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, Kiawah Island, December 1999.

[37] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proceedings of the 3rd USENIX symposium on Internet Technologies and Systems (USITS)*, March 2001.

[38] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[39] O. Spatscheck, J. Hansen, J. Hartman, and L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 2(8):146–157, 2000.

[40] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, and L. Zhang. Internet Engineering Task Force, RFC 2960: Stream Control Transmission Protocol, October 2000.

[41] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of the ACM SIGCOMM 2002 Conference on Communications Architectures and Protocols*, August 2002.

[42] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proceedings of the ACM Conference on Communications Architectures and Data Communication (SIGCOMM)*, September 1998.

[43] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active names: Flexible location and transport of wide-area resources. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, October 1999.

[44] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proceedings of 17th ACM Symposium on Operating System Principles (SOSP)*, December 1999.

[45] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *Proceedings of USENIX Technical Conference*, January 1997.

[46] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the NLANR Web Cache Workshop*, June 1997.

TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services

Jon Salz and Hari Balakrishnan
MIT Laboratory for Computer Science
{jsalz, hari}@lcs.mit.edu

Alex Snoeren
University of California, San Diego
snoeren@cs.ucsd.edu

Session-layer services for enhancing functionality and improving network performance are gaining in importance in the Internet. Examples of such services include connection multiplexing, congestion state sharing, application-level routing, mobility/migration support, and encryption. This paper describes TESLA, a transparent and extensible framework allowing session-layer services to be developed using a high-level flow-based abstraction. TESLA services can be deployed transparently using dynamic library interposition and can be composed by chaining event handlers in a graph structure. We show how TESLA can be used to implement several session-layer services including encryption, SOCKS, application-controlled routing, flow migration, and traffic rate shaping, all with acceptably low performance degradation.

1 Introduction

Modern network applications must meet several increasing demands for performance and enhanced functionality. Much current research is devoted to augmenting the transport-level functionality implemented by standard protocols as TCP and UDP. Examples abound:

- Setting up multiple connections between a source and destination to improve the throughput of a single logical data transfer (e.g., file transfers over high-speed networks where a single TCP connection alone does not provide adequate utilization [2, 15]).
- Sharing congestion information across connections sharing the same network path.
- Application-level routing, where applications route traffic in an overlay network to the final destination.
- End-to-end session migration for mobility across network disconnections.
- Encryption services for sealing or signing flows.
- General-purpose compression over low-bandwidth links.
- Traffic shaping and policing functions.

These examples illustrate the increasing importance of *session-layer services* in the Internet—services that operate on groups of flows between a source and destination, and produce resulting groups of flows using shared code and sometimes shared state.

Authors of new services such as these often implement enhanced functionality by augmenting the link, network, and transport layers, all of which are typically implemented in the kernel or in a shared, trusted intermediary [12]. While this model has sufficed in the past, we believe that a generalized, high-level framework for session-layer services would greatly ease their development and deployment. This paper argues that Internet end hosts can benefit from a *systematic* approach to developing session-layer services compared to the largely *ad-hoc* point approaches used today, and presents TESLA (a Transparent, Extensible Session Layer Architecture), a framework that facilitates the development of session-layer services like the ones mentioned above.

Our work with TESLA derives heavily from our own previous experience developing, debugging, and deploying a variety of Internet session-layer services. The earliest example is the Congestion Manager (CM) [4], which allows concurrent flows with a common source and destination to share congestion information, allocate available bandwidth, and adapt to changing network conditions. Other services include Resilient Overlay Networks (RON) [3], which provides application-layer routing in an overlay network, and the Migrate mobility architecture [23, 24], which preserves end-to-end communication across relocation and periods of disconnection.

Each these services was originally implemented at the kernel level, though it would be advantageous (for portability and ease of development and deployment) to have them available at the user level. Unfortunately this tends to be quite an intricate process. Not only must the implementation specify the internal logic, algorithms, and API, but considerable care must be taken handling the details of non-blocking and blocking sockets, interprocess communication, process management, and integrating the API with the application's event loop. The end result is that more programmer time and effort is spent setting up the session-layer plumbing than in the service's logic itself. Our frustrations with the development and implementation of these services at the user level were the prime motivation behind TESLA and led to three explicit design goals.

First, it became apparent to us that the standard BSD sockets API is not a convenient abstraction for programming session-layer services. Sockets can be duplicated and shared across processes; operations on them can be blocking or non-blocking on a descriptor-by-descriptor basis; and reads and writes can be multiplexed using several different APIs (e.g., select and poll). It is undesirable to require each service author to implement the entire sockets API. In response, TESLA exports a higher level of abstraction to session services, allowing them to operate on network flows (rather than simply socket descriptors) and treat flows as objects to be manipulated.

Second, there are many session services that are required *ex post facto*, often not originally thought of by the application developer but desired later by a user. For example, the ability to shape or police flows to conform to a specified peak rate is often useful, and being able to do so without kernel modifications is a deployment advantage. This requires the ability to configure session services transparent to the application. To do this, TESLA uses an old idea—dynamic library interposition [9]—taking advantage of the fact that most applications today on modern operating systems use dynamically linked libraries to gain access to kernel services. This does not, however, mean that TESLA session-layer services must be transparent. On the contrary, TESLA allows services to define APIs to be exported to enhanced applications.

Third, unlike traditional transport and network layer services, there is a great diversity in session services as the examples earlier in this section show. This implies that application developers can benefit from composing different available services to provide interesting new functionality. To facilitate this, TESLA arranges for session services to be written as event handlers, with a callback-oriented interface between handlers that are arranged in a graph structure in the system.

We argue that a generalized architecture for the development and deployment of session-layer functionality will significantly assist in the implementation and use of new network services. This paper describes the design and implementation of TESLA, a generic framework for development and deployment of session-layer services. TESLA consists of a set of C++ application program interfaces (APIs) specifying how to write these services, and an interposition agent that can be used to instantiate these services for use by existing applications.

2 Related Work

Today's commodity operating systems commonly allow the dynamic installation of network protocols on a system-wide or per-interface basis (e.g., Linux kernel modules and FreeBSD's netgraph), but these extensions can only be accessed by the super-user. Some operating systems, such as SPIN [5] and the Exokernel [12], push many operating system features (like network and file system access) out from the kernel into application-specific, user-configurable libraries, allowing ordinary users fine-grained control. Alternatively, extensions were developed for both operating systems to allow applications to define application-specific handlers that may be installed directly into the kernel (Plexus [14] and ASHs [27]).

Operating systems such as Scout [21] and *x*-kernel [16] were designed explicitly to support sophisticated network-based applications. In these systems, users may even redefine network or transport layer protocol functions in an application-specific fashion [6]. With TESLA, our goal is to bring some of the power of these systems to commodity operating systems in the context of session-layer services.

In contrast to highly platform-dependent systems such as U-Net [26] and Alpine [11], TESLA does not attempt to allow users to replace or modify the system's network stack. Instead, it focuses on allowing users to dynamically *extend* the protocol suite by dynamically composing additional end-to-end session-layer protocols on top of the existing transport- and network-layer protocols, achieving greater platform independence and usability.

TESLA's modular structure, comprising directed graphs of processing nodes, shares commonalities with a number of previous systems such as *x*-kernel, the Click modular router [19], and UNIX System V streams [22]. Unlike the transport and network protocols typically considered in these systems, however, we view session-layer protocols as an extension of the application itself rather than a system-wide resource. This ensures that they are subject to the same scheduling, protection, and resource constraints as the application, minimizing the amount of effort (and privileges) required to deploy services.

To avoid making changes to the operating system or to the application itself, TESLA transparently interposes itself between the application and the kernel, intercepting and modifying the interaction between the application and the system—acting as an *interposition agent* [18]. It uses dynamic library interposition [9] to modify the interaction between the application and the system. This technique is popular with user-level file systems such as IFS [10] and Ufo [1], and several libraries that provide specific, transparent network services such as SOCKS [20], Reliable Sockets [29], and Migrate [23, 24]. Each of these systems provides only a specific service, however, not an architecture usable by third parties.

Conductor [28] traps application network operations and transparently layers composable "adaptors" on TCP connections, but its focus is on optimizing flows' performance characteristics, not on providing arbitrary additional services. Similarly, Protocol Boosters [13] proposes interposing transparent agents between communication endpoints to improve performance over particular links (e.g., compression or forward error correction). While Protocol Boosters were originally implemented in the FreeBSD and Linux kernel, they are an excellent example of a service that could be implemented in a generic fashion using TESLA. Thain and Livny recently proposed Bypass, a dynamic-library based interposition toolkit for building split-execution agents commonly found in distributed systems [25]. However, because of their generality, none of these systems provides any assistance in building modular network services, particularly at the session layer. To the best of our knowledge, TESLA is the first interposition toolkit to specifically support generic session-layer network services.

3 Architecture

As discussed previously, many tools exist that, like TESLA, provide an interposition (or "shim") layer between applications and operating system kernels or libraries. However, TESLA raises the level of abstraction of programming for session-layer services. It does so by introducing a *flow handler* as the main object manipulated by session services. Each session service is implemented as an instantiation of a flow handler, and TESLA takes care of the plumbing required to allow session services to communicate with one another.

A *network flow* is a stream of bytes that all share the same logical source and destination (generally identified by source and destination IP addresses, source and destination port numbers, and transport protocol). Each flow handler takes as input a single network flow, and produces zero or more network flows as output. Flow handlers perform some particular operations or transformations on the byte stream, such as transparent flow migration, encryption, compression, etc.

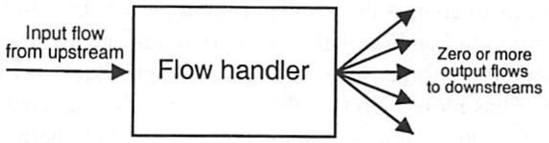


Figure 1: A flow handler takes as input one network flow and generates zero or more output flows.

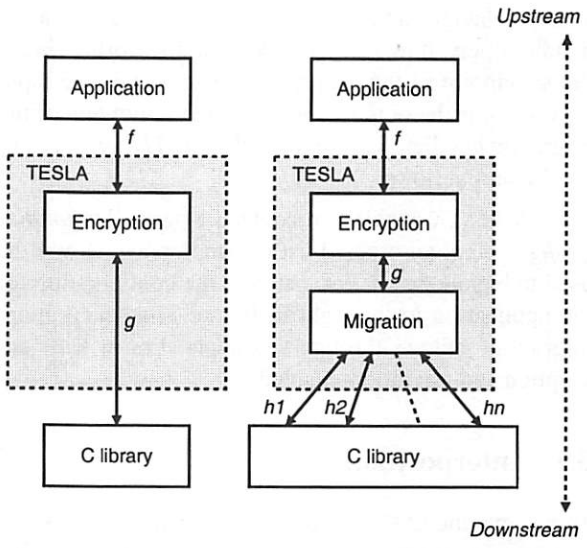


Figure 2: Two TESLA stacks. The encryption flow handler implements input flow f with output flow g. The migration flow handler implements input flow g with output flows $h_1 \dots h_n$.

A flow handler, illustrated in Figure 1, is explicitly defined and constructed to operate on only one *input flow* from an *upstream* handler (or end application), and is hence devoid of any demultiplexing operations. Conceptually, therefore, one might think of a flow handler as dealing with traffic corresponding to a single socket only (as opposed to an interposition layer coded from scratch, which must potentially deal with operations on *all* open file descriptors). A flow handler generates zero or more *output flows*, which map one-to-one to *downstream* handlers (or the network send routine).[1] While a flow handler always has one input flow, multiple flow handlers may coexist in a single process, so they may easily share global state. (We will expound on this point as we further discuss the TESLA architecture.)

The left stack in Figure 2 illustrates an instance of TESLA where stream encryption is the only enabled handler. While the application's I/O calls *appear* to be reading and writing plaintext to some flow f, in reality

TESLA intercepts these I/O calls and passes them to the encryption handler, which actually reads and writes ciphertext on some other flow g. The right stack in Figure 2 has more than two flows. f is the flow as viewed by the application, i.e., plaintext. g is the flow between the encryption handler and the migration handler, i.e., ciphertext. $h_1, h_2, \ldots, h_n$ are the n flows that a migration flow handler uses to implement flow g. (The migration handler initially opens flow h_1. When the host's network address changes and h_1 is disconnected, the migration handler opens flow h_2 to the peer, and so forth.) From the standpoint of the encryption handler, f is the input flow and g is the output flow. From the standpoint of the migration handler, g is the input flow and $h_1, h_2, \ldots, h_n$ are the output flows.

Each TESLA-enabled process has a particular *handler configuration*, an ordered list of handlers which will be used to handle flows. For instance, the configuration for the application to the right in Figure 2 is "encryption; migration." Flows through this application will be encryption and migration-enabled.

3.1 Interposition

To achieve the goal of application transparency, TESLA acts as an interposition agent at the C-library level. We provide a wrapper program, tesla, which sets up the environment (adding our libtesla.so shared library to LD_PRELOAD) and invokes an application, e.g.:

```
tesla +crypt -key=sec.gpg +migrate ftp mars
```

This would open an FTP connection to the host named mars, with encryption (using sec.gpg as the private key) and end-to-end migration enabled.

We refer to the libtesla.so library as the TESLA *stub*, since it contains the interposition agent but not any of the handlers themselves (as we will discuss later).

3.2 The flow_handler API

Every TESLA session service operates on flows and is implemented as a derived class of flow_handler, shown in Figure 3. To instantiate a downstream flow handler, a flow handler invokes its protected plumb method. TESLA handles plumb by instantiating handlers which appear after the current handler in the configuration.

For example, assume that TESLA is configured to perform compression, then encryption, then session migration on each flow. When the compression handler's constructor calls plumb, TESLA responds by instantiating the next downstream handler, namely the encryption handler; when its constructor in turn calls plumb, TESLA

```
class flow_handler {
  protected:
    flow_handler *plumb(int domain, int type);
    handler* const upstream;
    vector<handler*> downstream;

  public:
    // DOWNSTREAM methods
    virtual int connect(address);
    virtual int bind(address);
    virtual pair<flow_handler*, address> accept();
    virtual int close();
    virtual bool write(data);
    virtual int shutdown(bool r, bool w);
    virtual int listen(int backlog);
    virtual address getsockname();
    virtual address getpeername();
    virtual void may_avail(bool);
    virtual string getsockopt(...);
    virtual int setsockopt(...);
    virtual int ioctl(...);

    // UPSTREAM methods ('from' is a downstream flow)
    virtual void connected(flow_handler *from, bool success);
    virtual void accept_ready(flow_handler *from);
    virtual bool avail(flow_handler *from, data);
    virtual void may_write(flow_handler *from, bool may);
};
```

Figure 3: An excerpt from the flow_handler class definition. address and data are C++ wrapper classes for address and data buffers, respectively.

instantiates the next downstream handler, namely migration. Its plumb method creates a TESLA-internal handler to implement an actual TCP socket connection.

To send data to a downstream flow handler, a flow handler invokes the latter's write method, which in turn typically performs some processing and invokes its own downstream handler's write method. Downstream handlers communicate with upstream ones via callbacks, which are invoked to make events available to the upstream flow.

Flow handler methods are *asynchronous* and *event-driven*—each method call must return immediately, without blocking.

3.3 Handler method semantics

Many of the virtual flow_handler methods presented in Figure 3—write, connect, getpeername, etc.—have semantics similar to the corresponding non-blocking function calls in the C library. (A handler class may override any method to implement it specially, i.e., to change the behavior of the flow according to the session service the handler provides.) These methods are invoked by a handler's input flow. Since, in general, handlers will propagate these messages to their output flows (i.e., downstream), these methods are called *downstream methods* and can be thought of as providing an abstract flow service to the upstream handler.

(One downstream method has no C-library analogue: may_avail informs the handler whether or not it may make any more data available to its upstream handler. We further discuss this method in Section 3.5.)

In contrast the final four methods are invoked by the handler's *output flows*; each has a from argument identifying which downstream handler is invoking the method. Since typically handlers will propagate these messages to their input flow (i.e., upstream), these methods are called *upstream methods* and can be thought of callbacks which are invoked by the upstream handler.

- connected is invoked when a connection request (i.e., a connect method invocation) on a downstream has completed. The argument is true if the request succeeded or false if not.
- accept_ready is invoked when listen has been called and a remote host attempts to establish a connection. Typically a flow handler will respond to this by calling the downstream flow's accept method to accept the connection.
- avail is invoked when a downstream handler has received bytes. It returns false if the upstream handler is no longer able to receive bytes (i.e., the connection has shut down for reading).
- may_write, analogous to may_avail, informs the handler whether or not it may write any more data downstream. This method is further discussed in Section 3.5.

Many handlers, such as the encryption handler, perform only simple processing on the data flow, have exactly one output flow for one input flow, and do not modify the semantics of other methods such as connect or accept. For this reason we provide a default implementation of each method which simply propagates method calls to the downstream handler (in the case of downstream methods) or the upstream handler (in the case of upstream methods).

We further discuss the input and output primitives, timers, and blocking in the next few sections.

3.4 Input/output semantics

Since data input and output are the two fundamental operations on a flow, we shall describe them in more detail. The write method call writes bytes to the flow. It returns a boolean indicating success, unlike the C library's write call which returns a number of bytes or an EAGAIN error message (we will explain this design decision shortly). Our interface lacks a read method; rather, we use a callback, avail, which is invoked by a downstream handler whenever bytes are available. The upstream handler handles the bytes immediately, generally by performing some processing on the bytes and passing them to its own upstream handler.

A key difference between flow_handler semantics and the C library's I/O semantics is that, in flow handlers, writes and avails are *guaranteed to complete*. In particular, a write or avail call returns false, indicating failure, only when it will never again be possible to write to or receive bytes from the flow (similar to the C library returning 0 for write or read). Contrast this to the C library's write and read function calls which (in non-blocking mode) may return an EAGAIN error, requiring the caller to retry the operation later. Our semantics make handler implementation considerably simpler, since handlers do not need to worry about handling the common but difficult situation where a downstream handler is unable to accept all the bytes it needs to write.

Consider the simple case where an encryption handler receives a write request from an upstream handler, performs stream encryption on the bytes (updating its state), and then attempts to write the encrypted data to the downstream handler. If the downstream handler could return EAGAIN, the handler must buffer the unwritten encrypted bytes, since by updating its stream encryption state the handler has "committed" to accepting the bytes from the upstream handler. Thus the handler would have to maintain a ring buffer for the unwritten bytes and register a callback when the output flow is available for writing.

Our approach (guaranteed completion) benefits from the observation that given upstream and downstream handlers that support guaranteed completion, it is easy to write a handler to support guaranteed completion.[2] This is an inductive argument, of course—there must be some blocking mechanism in the system, i.e., completion cannot be guaranteed everywhere. Our decision to impose guaranteed completion on handlers isolates the complexity of blocking in the implementation of TESLA itself (as described in Section 5.1), relieving handler authors of having to deal with multiplexing between different flows (e.g., with select).[3]

3.5 Timers and flow control

As we have mentioned before, TESLA handlers are event-driven, hence all flow_handler methods must return immediately. Clearly, however, some flow control mechanism is required in flow handlers—otherwise, in an application where the network is the bottleneck, the application could submit data to TESLA faster than TESLA could ship data off to the network, requiring TESLA to buffer a potentially unbounded amount of data. Similarly, if the application were the bottleneck, TESLA would be required to buffer all the data flowing upstream from the network until the application was ready to process it.

A flow handler may signal its input flow to stop sending it data, i.e., stop invoking its write method, by invoking may_write(false); it can later restart the handler by invoking may_write(true). Likewise, a handler may throttle an output flow, preventing it from calling avail, by invoking may_avail(false). Flow handlers are *required* to respect may_avail and may_write requests from downstream and upstream handlers. This does not complicate our guaranteed-completion semantics, since a handler which (like most) lacks a buffer in which to store partially-processed data can simply propagate may_writes upstream and may_avails downstream to avoid receiving data that it cannot handle.

A handler may need to register a time-based callback, e.g., to re-enable data flow from its upstream or downstream handlers after a certain amount of time has passed. For this we provide a timer facility allowing handlers to instruct the TESLA event loop to invoke a callback at a particular time.

3.6 Handler-specific services

Each flow handler exposes (by definition) the flow_handler interface, but some handlers may need to provide additional services to applications that wish to support them specifically (but still operate properly if TESLA or the particular handler is not available or not enabled). For instance, the end-to-end migration handler can "freeze" an application upon network disconnection, preserving the process's state and re-creating it upon reconnection. To enable this functionality we introduced the ioctl method to flow_handler. A "Migrate-aware" application (or, in the general case, a "TESLA-aware" application) may use the ts_ioctl macro to send a control message to a handler and receive a response:

```
struct freeze_params_t params = { ... };
struct freeze_return_t ret;
int ret = ts_ioctl(fd, "migrate", MIGRATE_FREEZE,
    &params, sizeof params, &ret, sizeof ret);
```

We also provide a mechanism for handlers to send events to the application asynchronously; e.g., the migration handler can be configured to notify the application when a flow has been migrated between IP addresses.

3.7 Process management

Flow handlers comprise executable code and runtime state, and in designing TESLA there were two obvious choices regarding the context within which the flow handler code would run. The simplest approach would place flow handlers directly within the address space of application processes; TESLA would delegate invocations of POSIX I/O routines to the appropriate flow_handler methods. Alternatively, flow handlers could execute in separate processes from the application; TESLA would manage the communication between application processes and flow handler processes.

The former, simpler approach would not require any interprocess communication or context switching, minimizing performance degradation. Furthermore, this approach would ensure that each flow handler has the same process priority as the application using it, and that there are no inter-handler protection problems to worry about.

Unfortunately, this approach proves problematic in several important cases. First, some session services (e.g., shared congestion management as in CM) require state to be shared across flows that may belong to different application processes, and making each flow handler run linked with the application would greatly complicate the ability to share session-layer state across them. Second, significant difficulties arise if the application process shares its flows (i.e., the file descriptors corresponding to the flows) with another process, either by creating a child process through fork or through file descriptor passing. Ensuring the proper sharing semantics becomes difficult and expensive. A fork results in two identical TESLA instances running in the parent and child, making it rather cumbersome to ensure that they coordinate correctly. Furthermore, maintaining the correct semantics of asynchronous I/O calls like select and poll is very difficult in this model.

For these reasons, we choose to separate the context in which TESLA flow handlers run from the application processes that generate the corresponding flows. When an application is invoked through the tesla wrapper (and hence linked against the stub), the stub library creates or connects to a *master process*, a process dedicated to executing handlers.

Each master process has its own handler configuration (determined by the user, as we shall describe later). When the application establishes a connection, the master process determines whether the configuration contains any applicable handlers. If so, the master instantiates the applicable handlers and links them together in what we call a TESLA *instance*.

The application and master process communicate via

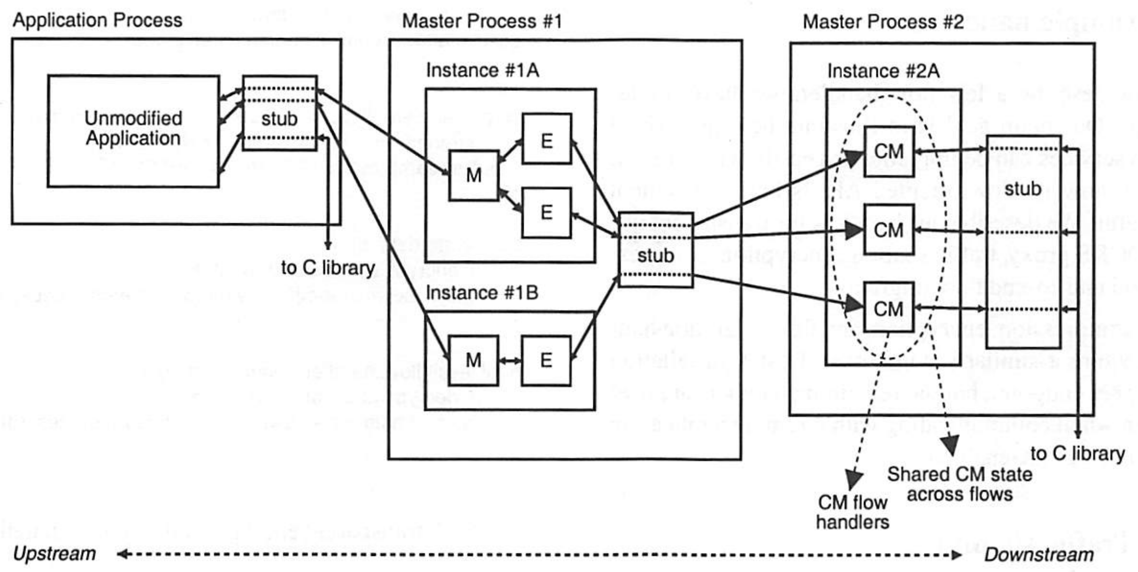


Figure 4: Possible interprocess data flow for an application running under TESLA. M is the migration handler, E is encryption, and CM is the congestion manager handler.

a TESLA-internal socket for each application-level flow, and all actual network operations are performed by the master process. When the application invokes a socket operation, TESLA traps it and converts it to a message which is transmitted to the master process via the master socket. The master process returns an immediate response (this is possible since all TESLA handler operations are non-blocking, i.e., return immediately).

Master processes are forked from TESLA-enabled applications and are therefore linked against the TESLA stub also. This allows TESLA instances to be chained, as in Figure 4: an application-level flow may be connected to an instance in a first master process, which is in turn connected via its stub to an instance in another master process. Chaining enables some handlers to run in a protected, user-specific context, whereas handlers that require a system-scope (like system-wide congestion management) can run in a privileged, system-wide context.

3.8 Security considerations

Like most software components, flow handlers can potentially wreak havoc within their protection context if they are malicious or buggy. Our chaining approach allows flow handlers to run within the protection context of the user to minimize the damage they can do; but in any case, flow handlers must be trusted within their protection contexts. Since a malicious flow handler could potentially sabotage all connections in the same master process, any flow handlers in master processes intended to have system-wide scope (such as a Congestion Manager handler) must be trusted.

In general, TESLA support for setuid and setgid applications, which assume the protection context of the binary iself rather than the user who invokes them, is a tricky affair: one cannot trust a user-provided handler to behave properly within the binary's protection context. Even if the master process (and hence the flow handlers) run within the protection context of the user, user-provided handlers might still modify network semantics to change the behavior of the application. Consider, for example, a setuid binary which assumes root privileges, performs a lookup over the network to determine whether the user is a system administrator and, if so, allows the user to perform an administrative operation. If a malicious user provides flow handlers that spoof a positive response for the lookup, the binary may be tricked into granting him or her administrative privileges.

Nevertheless, many widely-used applications (e.g., ssh) are setuid, so we do require some way to support them. We can make the tesla wrapper setuid root, so that it is allowed to add libtesla.so to the LD_PRELOAD environment variable even for setuid applications. The tesla wrapper then invokes the requested binary, linked against libtesla.so, with the appropriate privileges. libtesla.so creates a master process and begins instantiating handlers *only* once the process resets its effective user ID to the user's real user ID, as is the case with applications such as ssh. This ensures that only network connections within the user's protection context can be affected (maliciously or otherwise) by handler code.

4 Example handlers

We now describe a few flow handlers we have implemented. Our main goal is to illustrate how non-trivial session services can be implemented easily with TESLA, showing how its flow-oriented API is both convenient and useful. We describe our handlers for transparent use of a SOCKS proxy, traffic shaping, encryption, compression, and end-to-end flow migration.

The compression, encryption, and flow migration handlers require a similarly configured TESLA installation on the peer endpoint, but the remaining handlers are useful even when communicating with a remote application that is not TESLA-enabled.

4.1 Traffic shaping

It is often useful to be able to limit the maximum throughput of a TCP connection. For example, one might want prevent a background network operation such as mirroring a large software distribution to impact network performance. TESLA allows us to provide a generic, user-level traffic-shaping service as an alternative to building shaping directly into an application (as in the rsync --bwlimit option) or using an OS-level packet-filtering service (which would require special setup and superuser privileges).

The traffic shaper keeps count of the number of bytes it has read and written during the current 100-millisecond timeslice. If a write request would exceed the outbound limit for the timeslice, then the portion of the data which could not be written is saved, and the upstream handler is throttled via may_write. A timer is created to notify the shaper at the end of the current timeslice so it can continue writing and unthrottle the upstream handler when appropriate. Similarly, once the inbound bytes-per-timeslice limit is met, any remaining data provided by avail is buffered, downstream flows are throttled, and a timer is registered to continue reading later.

We have also developed latency_handler, which delays each byte supplied to or by the network for a user-configurable amount of time (maintaining this data in an internal ring buffer). This handler is useful for simulating the effects of network latency on existing applications.

4.2 SOCKS and application-level routing

Our SOCKS handler is functionally similar to existing transparent SOCKS libraries [7, 17], although its implementation is significantly simpler. Our handler overrides the connect handler to establish a connection to a proxy server, rather than a connection directly to the requested host. When its connected method is invoked, it does not pass it upstream, but rather negotiates the authentication mechanism with the server and then passes it the actual destination address, as specified by the SOCKS protocol.

```
class crypt_handler : public flow_handler {
    des3_cfb64_stream in_stream, out_stream;

  public:
    crypt_handler(init_context& ctxt) : flow_handler(ctxt),
      in_stream(des3_cfb64_stream::ENCRYPT),
      out_stream(des3_cfb64_stream::DECRYPT)
    {}

    bool write(data d) {
        // encrypt and pass downstream
        return downstream[0]->write(out_stream.process(d));
    }

    bool avail(flow_handler *from, data d) {
        // decrypt and pass upstream
        return upstream->avail(this, in_stream.process(d));
    }
};
```

Figure 5: A transparent encryption/decryption handler.

If the SOCKS server indicates that it was able to establish the connection with the remote host, then the SOCKS handler invokes connected(true) on its upstream handler; if not, it invokes connected(false). The upstream handler, of course, is never aware that the connection is physically to the SOCKS server (as opposed to the destination address originally provided to connect).

We utilize the SOCKS server to provide transparent support for Resilient Overlay Networks [3], or RON, an architecture allowing a group of hosts to route around failures or inefficiencies in a network. We provide a ron_handler to allow a user to connect to a remote host transparently through a RON. Each RON server exports a SOCKS interface, so ron_handler can use the same mechanism as socks_handler to connect to a RON host as a proxy and open an indirect connection to the remote host. In the future, ron_handler may also utilize the end-to-end migration of migrate_handler (presented below in Section 4.4) to enable RON to hand off the proxy connection to a different node if it discover a more efficient route from the client to the peer.

4.3 Encryption and compression

crypt_handler is a triple-DES encryption handler for TCP streams. We use OpenSSL [8] to provide the DES implementation. Figure 5 shows how crypt_handler uses the TESLA flow handler API. Note how simple the handler is to write: we merely provide alternative implementations for the write and avail methods, routing data first through a DES encryption or decryption step (des3_cfb64_stream, a C++ wrapper we have written for OpenSSL functionality).

Our compression handler is very similar: we merely wrote a wrapper class (analogous to des3_cfb64_stream) for the freely available zlib stream compression library.

4.4 Session migration

We have used TESLA to implement transparent support in the Migrate session-layer mobility service [23]. In transparent mode, Migrate preserves open network connections across changes of address or periods of disconnection. Since TCP connections are bound to precisely one remote endpoint and do not survive periods of disconnection in general, Migrate must synthesize a logical flow out of possibly multiple physical connections (a new connection must be established each time either endpoint moves or reconnects). Further, since data may be lost upon connection failure, Migrate must double-buffer in-flight data for possible re-transmission.

This basic functionality is straightforward to provide in TESLA. We simply create a handler that splices its input flow to an output flow and conceals any mobility events from the application by automatically initiating a new output flow using the new endpoint locations. The handler stores a copy of all outgoing data locally in a ring buffer and re-transmits any lost bytes after re-establishing connectivity on a new output flow. Incoming data is trickier: because received data may not have been read by the application before a mobility event occurs, Migrate must instantiate a new flow immediately after movement but continue to supply the buffered data from the previous flow to the application until it is completely consumed. Only then will the handler begin delivering data received on the subsequent flow(s).

Note that because Migrate wishes to conceal mobility when operating in transparent mode, it is important that the handler be able to override normal signaling mechanisms. In particular, it must intercept connection failure messages (the "connection reset" messages typically experienced during long periods of disconnection) and prevent them from reaching the application, instead taking appropriate action to manage and conceal the changes in endpoints. In doing so, the handler also overrides the getsockname() and getpeername() calls to return the original endpoints, irrespective of the current location.

5 Implementation

The TESLA stub consists largely of *wrapper functions* for sockets API functions in the C library. When an application creates a socket, TESLA intercepts the socket library call and sends a message to the master process, inquiring whether the master has any handlers registered for the requested domain and type. If not, the master returns a negative acknowledgement and the application process simply uses the socket system call to create and return the request socket.

If, on the other hand, there is a handler registered for the requested domain and type, the master process instantiates the handler class. Once the handler's constructor returns, the master process creates a pair of connected UNIX-domain sockets, one retained in the master and the other passed to the application process. The application process notes the received filehandle (remembering that it is a TESLA-wrapped filehandle) and returns it as result of the socket call.

Later, when the application invokes a socket API call, such as connect or getsockname, on a TESLA-wrapped filehandle, the wrapper function informs the master process, which invokes the corresponding method on the handler object for that flow. The master process returns this result to the stub, which returns the result to the application.

In general the TESLA stub does not need to specially handle reads, writes, or multiplexing on wrapped sockets. The master process and application process share a socket for each application-level flow, so to handle a read, write, or select, whether blocking or non-blocking, the application process merely uses the corresponding unwrapped C library call. Even if the operation blocks, the master process can continue handling I/O while the application process waits.

Implementing fork, dup, and dup2 is quite simple. Having a single flow available to more than one application process, or as more than one filehandle, presents no problem: application processes can use each copy of the filehandle as usual. Once all copies of the filehandle are shut down, the master process can simply detect via a signal that the filehandle has closed and invokes the handler's close method.

5.1 top_handler and bottom_handler

We have described at length the interface between handlers, but we have not yet discussed how precisely handlers receives data and events from the application. We have stated that every handler has an upstream flow which invokes its methods such as connect, write, etc.; the upstream flow of the *top-most* handler for each flow (e.g., the encryption handler in Figure 2) is a special flow handler called top_handler.

When a TESLA master's event loop detects bytes ready to deliver to a particular flow via the application/master-process socket pair which exists for that flow, it invokes the write method of top_handler, which passes the write call on to the first real flow handler (e.g., encryption). Similarly, when a connect, bind, listen, or accept message appears on the master socket, the TESLA event loop invokes the corresponding method of the top_handler for that flow.

Similarly, each flow handler at the bottom of the TESLA stack has a bottom_handler as its downstream. For each non-callback method—i.e., connect, write, etc.—bottom_handler actually performs the corresponding network operation.

top_handlers and bottom_handlers maintain buffers, in case a handler writes data to the application or network, but the underlying system buffer is full (i.e., sending data asynchronously to the application or network results in an EAGAIN condition). If this occurs in a top_handler, the top_handler requests via may_avail that its downstream flow stop sending it data using avail; similarly, if this occurs in a bottom_handler, it requests via may_write that its upstream flow stop sending it data using write.

5.2 Performance

When adding a network service to an application, we may expect to incur one or both of two kinds of performance penalties. First, the service's algorithm, e.g., encryption or compression, may require computational resources. Second, the interposition mechanism, if any, may introduce overhead such as additional memory copies, interprocess communication, scheduling contention, etc., depending on its implementation. We refer to these two kinds of performance penalty as *algorithmic* and *architectural*, respectively.

If the service is implemented directly within the application, e.g., via an application-specific input/output indirection layer, then the architectural overhead is probably minimal and most of the overhead is likely to be algorithmic. Services implemented entirely within the operating system kernel are also likely to impose little in the way of architectural overhead. However, adding an interposition agent like TESLA outside the contexts of both the application and the operating system may add significant architectural overhead.

To analyze TESLA's architectural overhead, we constructed several simple tests to test how using TESLA affects latency and throughput. We compare TESLA's performance to that of a tunneling, or proxy, agent, where the application explicitly tunnels flows through a local proxy server, and to an application-internal agent. Although we do not consider it here, we expect an in-kernel service implementation would perform similarly to the application-internal agent.

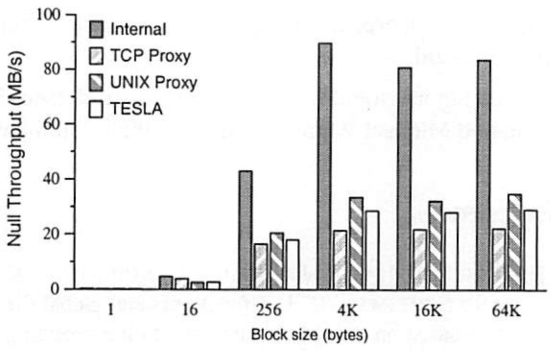
We provide two benchmarks. In bandwidth(s), a client establishes a TCP connection to a server and reads data into a s-byte buffer until the connection closes, measuring the number of bytes per second received. (The server writes data s bytes at a time as well.) In latency, a client connects to a server, sends it a single byte, waits for an 1-byte response, and so forth, measuring the number of round-trip volleys per second.

We run bandwidth and latency under several configurations:

1. The benchmark program running (a) unmodified and (b) with triple-DES encryption performed directly within the benchmark application.
2. The benchmark program, with each connection passing through TCP proxy servers both on the client and server hosts. In (a), the proxy servers perform no processing on the data; in (b) the proxy servers encrypt and decrypt traffic between the two hosts.
3. Same as 2(a) and 2(b), except with proxy servers listening on UNIX-domain sockets rather than TCP sockets. This is similar to the way TESLA works internally: the proxy server corresponds to the TESLA master process.
4. The benchmark program running under TESLA with (a) the dummy handler enabled or (b) the crypt handler enabled. (dummy is a simple no-op handler that simply passes data through.)

We can think of the (a) configurations as providing a completely trivial transparent service, an "identity service" with no algorithm (and hence no algorithmic overhead) at all. Comparing benchmark 1(a) with the other (a) configurations isolates the architectural overhead of a typical proxy server (i.e., an extra pair of socket streams through which all data must flow) and the architectural overhead of TESLA (i.e., an extra pair of socket streams plus any overhead incurred by the master processes). Comparing the (a) benchmarks, shown on the left of Figure 6, with the corresponding (b) benchmarks, shown on the right, isolates the algorithmic overhead of the service.

To eliminate the network as a bottleneck, we first ran the bandwidth test on a single host (a 500-MHz Pentium III running Linux 2.4.18) with the client and server connecting over the loopback interface. Figure 6 shows these results. Here the increased demand of interprocess communication is apparent. For large block sizes (which we would expect in bandwidth-intensive applications), introducing TCP proxies, and hence tripling the number



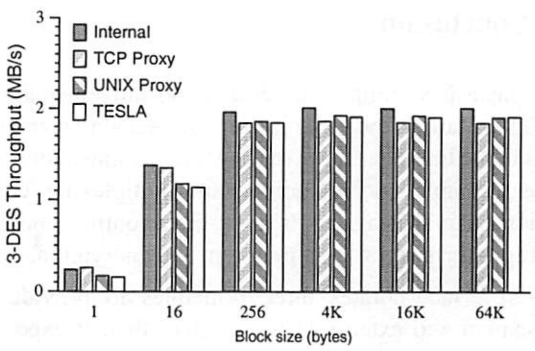


Figure 6: Results of the bandwidth test on a loopback interface.

of TCP socket streams involved, reduces the throughput by nearly two thirds. TESLA does a little better, since it uses UNIX-domain sockets instead of TCP sockets.

In contrast, Figure 7 shows the results of running bandwidth with the client and server (both 500-MHz Pentium IIIs) connected via a 100-BaseT network.[4] Here neither TESLA nor a proxy server incurs a significant throughput penalty. For reasonably large block sizes either the network (at about 89 Mb/s, or 11 MB/s) or the encryption algorithm (at about 3 MB/s) becomes the bottleneck; for small block sizes the high frequency of small application reads and writes is the bottleneck.

We conclude that on relatively slow machines and very fast networks, TESLA—or any other IPC-intensive interposition mechanism—may cause a decrease in peak throughput, but when used over typical networks, or when used to implement nontrivial network services, TESLA causes little or no throughput reduction.

The latency benchmark yields different results, as illustrated in Figure 8: since data rates never exceed a few kilobytes per second, the computational overhead of the algorithm itself is negligible and any slowdown is due exclusively to the architectural overhead. We see almost no difference between the speed of the trivial service and the nontrivial service. Introducing a TCP proxy server on each host incurs a noticeable performance penalty, since data must now flow over a total of three streams (rather than one) from client application to server application. TESLA does a little better, as it uses UNIX-domain sockets rather than TCP sockets to transport data between the applications and the TESLA masters; under Linux, UNIX-domain sockets appear to have a lower latency than the TCP sockets used by the proxy server. We consider this performance hit quite acceptable: TESLA increases the end-to-end latency by only tens of microseconds (4000 round-trips per second *versus* 7000), whereas network latencies are typically on the order of milliseconds or tens of milliseconds.

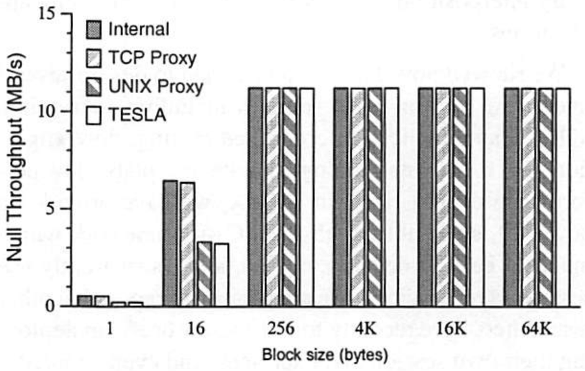


Figure 7: Results of the bandwidth test on a 100-BaseT network. Only the null test is shown; triple-DES performance was similar to performance on a loopback interface (the right graph of Figure 6).

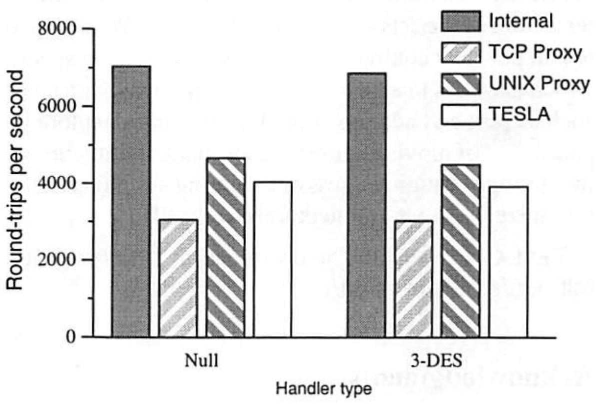


Figure 8: Results of the latency test over a 100-BaseT network. The block size is one byte.

6 Conclusion

This paper has outlined the design and implementation of TESLA, a framework to implement session layer services in the Internet. These services are gaining in importance; examples include connection multiplexing, congestion state sharing, application-level routing, mobility/migration support, compression, and encryption.

TESLA incorporates three principles to provide a transparent and extensible framework: first, it exposes network flows as the object manipulated by session services, which are written as flow handlers without dealing with socket descriptors; second, it maps handlers to processes in a way that enables both sharing and protection; and third, it can be configured using dynamic library interposition, thereby being transparent to end applications.

We showed how TESLA can be used to design several interesting session layer services including encryption, SOCKS and application-controlled routing, flow migration, and traffic rate shaping, all with acceptably low performance degradation. In TESLA, we have provided a powerful, extensible, high-level C++ framework which makes it easy to develop, deploy, and transparently use session-layer services. We are pleased to report that other researchers have recently found TESLA useful in deploying their own session layer services, and even adopted it as a teaching tool.

Currently, to use TESLA services such as compression, encryption, and migration that require TESLA support on both endpoints, the client and server must use identical configurations of TESLA, i.e., invoke the tesla wrapper with the same handlers specified on the command line. We plan to add a negotiation mechanism so that once mutual TESLA support is detected, the endpoint can dynamically determine the configuration based on the intersection of the sets of supported handlers. We also plan to add per-flow configuration so that the user can specify which handlers to apply to flows based on flow attributes such as port and address. Finally, we plan to explore the possibility of moving some handler functionality directly into the application process or operating system kernel to minimize TESLA's architectural overhead.

TESLA is available for download at http://nms.lcs.mit.edu/software/tesla/.

Acknowledgments

This work was funded by NTT Inc. under the NTT-MIT research collaboration, by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, by Intel Corp., and by IBM Corp. under a university faculty award.

The authors would like to thank Dave Andersen, Stan Rost, and Michael Walfish of MIT for their comments.

References

[1] ALEXANDROV, A. D., IBEL, M., SCHAUSER, K. E., AND SCHEIMAN, C. J. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM TOCS 16*, 3 (Aug. 1998), 207–233.

[2] ALLMAN, M., KRUSE, H., AND OSTERMANN, S. An application-level solution to TCP's inefficiencies. In *Proc. WOSBIS '96* (Nov. 1996).

[3] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. T. Resilient overlay networks. In *Proc. ACM SOSP '01* (Oct. 2001), pp. 131–145.

[4] BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. An integrated congestion management architecture for Internet hosts. In *Proc. ACM SIGCOMM '99* (Sept. 1999), pp. 175–187.

[5] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM SOSP '95* (Dec. 1995), pp. 267–284.

[6] BHATTI, N. T., AND SCHLICHTING, R. D. A system for constructing configurable high-level protocols. In *Proc. ACM SIGCOMM '95* (Aug. 1995), pp. 138–150.

[7] CLOWES, S. tsocks: A transparent SOCKS proxying library. http://tsocks.sourceforge.net/.

[8] COX, M. J., ENGELSCHALL, R. S., HENSON, S., AND LAURIE, B. Openssl: The open source toolkit for SSL/TLS. http://www.openssl.org/.

[9] CURRY, T. W. Profiling and tracing dynamic library usage via interposition. In *Proc. Summer USENIX '94* (June 1994), pp. 267–278.

[10] EGGERT, P. R., AND PARKER, D. S. File systems in user space. In *Proc. Winter USENIX '93* (Jan. 1993), pp. 229–240.

[11] ELY, D., SAVAGE, S., AND WETHERALL, D. Alpine: A user-level infrastructure for network protocol development. In *Proc. 3rd USITS* (Mar. 2001), pp. 171–183.

[12] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proc. ACM SOSP '95* (Dec. 1995), pp. 251–266.

[13] FELDMEIER, D. C., MCAULEY, A. J., SMITH, J. M., BAKIN, D. S., MARCUS, W. S., AND RALEIGH, T. M. Protocol boosters. *IEEE JSAC 16*, 3 (Apr. 1998), 437–444.

[14] FIUCZYNSKI, M. E., AND BERSHAD, B. N. An extensible protocol architecture for application-specific networking. In *Proc. USENIX '96* (Jan. 1996), pp. 55–64.

[15] GEVROS, P., RISSO, F., AND KIRSTEIN, P. Analysis of a method for differential TCP service. In *Proc. IEEE GLOBECOM '99* (Dec. 1999), pp. 1699–1708.

[16] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–76.

[17] INFERNO NETTVERK A/S. Dante: A free SOCKS implementation. http://www.inet.no/dante/.

[18] JONES, M. B. Interposition agents: Transparently interposing user code at the system interface. In *Proc. ACM SOSP '93* (Dec. 1993), pp. 80–93.

[19] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM TOCS 18*, 3 (Aug. 2000), 263–297.

[20] LEECH, M., GANIS, M., LEE, Y., KURIS, R., KOBLAS, D., AND JONES, L. *SOCKS Protocol Version 5*. IETF, Mar. 1996. RFC 1928.

[21] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the Scout operating system. In *Proc. OSDI '96* (Oct. 1996), pp. 153–167.

[22] RICHIE, D. M. A stream input-output system. *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct. 1984), 1897–1910.

[23] SNOEREN, A. C. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, Dec. 2002.

[24] SNOEREN, A. C., BALAKRISHNAN, H., AND KAASHOEK, M. F. Reconsidering Internet Mobility. In *Proc. HotOS-VIII* (May 2001), pp. 41–46.

[25] THAIN, D., AND LIVNY, M. Multiple bypass: Interposition agents for distributed computing. *Cluster Computing 4*, 1 (Mar. 2001), 39–47.

[26] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. ACM SOSP '95* (Dec. 1995), pp. 40–53.

[27] WALLACH, D. A., ENGLER, D. R., AND KAASHOEK, M. F. ASHs: Application-specific handlers for high-performance messaging. In *Proc. ACM SIGCOMM '96* (Aug. 1996), pp. 40–52.

[28] YARVIS, M., REIHER, P., AND POPEK, G. J. Conductor: A framework for distributed adaptation. In *Proc. HotOS-VII* (Mar. 1999), pp. 44–51.

[29] ZANDY, V. C., AND MILLER, B. P. Reliable network connections. In *Proc. ACM/IEEE Mobicom '02* (Atlanta, Georgia, Sept. 2002), pp. 95–106.

Notes

[1]It is important not to take the terms upstream and downstream too literally in terms of the flow of actual bytes; rather, think of TESLA as the session layer on the canonical network stack, with upstream handlers placed closer to the presentation or application layer and downstream handlers closer to the transport layer.

[2]The inverse—"given upstream and downstream handlers that do not support guaranteed completion, it is easy to write a handler that does not support guaranteed completion"—is not true, for the reason described in the previous paragraph.

[3]The guaranteed-completion semantics we discuss in this section apply to the TESLA flow_handler API only, *not* to standard socket I/O functions (read, write, etc.). TESLA makes sure that the semantics of POSIX I/O functions remain unchanged, for transparency's sake.

[4] Note the anomaly with small block sizes: on our test machines, using a UNIX-domain socket to connect the application to the intermediary process (whether a proxy server or the TESLA master) incurs a severe performance hit. Reconfiguring TESLA to use TCP sockets internally (rather than UNIX-domain sockets) increases TESLA's performance to that of the TCP proxy server. Since this anomaly does not seem specific to TESLA, and occurs only in a somewhat unrealistic situation—a high-bandwidth application using a 1- or 16-byte block size—we do not examine it further.

Scriptroute: A Public Internet Measurement Facility

Neil Spring, David Wetherall and Tom Anderson
{nspring,djw,tom}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

Abstract

We present Scriptroute, a system that allows ordinary Internet users to conduct network measurements from remote vantage points. We seek to combine the flexibility found in dedicated measurement testbeds such as NIMI with the general accessibility and popularity of Web-based public traceroute servers. To use Scriptroute, clients use DNS to discover measurement servers and then submit a measurement script for execution in a sandboxed, resource-limited environment. The servers ensure that the script does not expose the network to attack by applying source- and destination-specific filters and security checks, and by rate-limiting traffic.

Scriptroute code is publicly available and has been deployed on the PlanetLab testbed of 42 sites. As proof-of-concept, we have used it both to create RPT, a tool for measuring routing trees toward a destination, and to repeat the experiment used to evaluate GNP, a recently proposed Internet distance estimation technique. We find that our system is flexible enough to implement a variety of measurement tools despite its security restrictions, that access to many remote vantage points makes the system valuable, and that scripting is an apt choice for expressing and combining measurement tasks.

1 Introduction

The ability to measure the Internet is of widespread value for diagnosing connectivity problems and understanding Internet topology [20, 53], routing [35, 54] and performance [3, 51]. This paper considers a simple question: what is the right architecture for a generally available network measurement facility?

Existing systems such as NIMI [45] provide much of the needed functionality, but not all. These research systems provide the advantages of dedicated hardware that can be used for a wide range of network measurements. In return, users must possess credentials or an account, which creates a barrier that limits access to a community of users trusted by the administrator. Thus these systems do not help unaffiliated users like a network operator trying to debug poor network performance.

The popularity of Web-accessible traceroute servers offers a different solution. Several hundred public traceroute servers are available, constituting the largest de facto Internet measurement facility. These servers are typically used to debug two-way connectivity problems, providing indirect benefit to the traceroute server host. They are also easy to secure, because they provide only limited functionality and local administrators retain control to deny access to abusive users. As a result, many network operators now contribute traceroute servers.

However, traceroute servers provide limited functionality – only a hop-by-hop TTL test – and have significant drawbacks when used as a measurement system. They are difficult to coordinate because they were not designed with programmed access in mind. They can be highly inefficient for some applications, such as our RPT tool described in Section 5.1. More importantly, there are many non-intrusive tests of path properties that are not supported by traceroute servers: tests for path MTU [17], available bandwidth [27, 55], capacity [33, 49], queuing and congestion [5], and reordering [4]. In short, it is clear that a much richer diagnostic and measurement capability would be possible with a general-purpose tool.

Our goal is to combine the best of both worlds: the flexibility to run a wide variety of different measurement tools with the general availability of traceroute servers. We begin with the safety properties of traceroute servers: we design the system to prevent misuse, even at the cost of disallowing some kinds of useful measurements. Our thesis is that even within the context of a carefully controlled interface, we can provide more functionality than is currently provided by traceroute servers. We hope to succeed to the point where administrators will find it to their advantage to host a Scriptroute server in place of their current traceroute server.

We call our system Scriptroute. We use scripting to facilitate the implementation of measurement tools and the coordination of measurements across servers. For example, traceroute can be expressed in Scriptroute in tens of lines of code (Section 3), instead of hundreds; and tasks can be combined across servers in hundreds of lines (Section 5) instead of the thousands required in a previous project [53]. For security, we use sandboxing and local control over resources to protect the measurement host, and rate-limiting and filters that block known attacks to protect the network. Further, because network measurements often send probe traffic to random Internet hosts and administrators sometimes mistake measurement traffic for an attack, we provide a mechanism for sites to block unwanted measurement traffic.

While none of the pieces of the design are particularly new (e.g., others have sandboxed foreign code [18, 23]), we believe that the result is novel and can substantially improve our ability to make safe, flexible remote measurements. Further, part of our goal is to spark a debate as to how a network measurement facility should be architected. Because we could have made different design choices, we see our system as only one design point in the space of network measurement service architectures. More broadly, given the rising popularity of various forms of widely accessible remote execution facilities, e.g., Akamai, .NET, and seti@home, our work provides an example of how to balance the tradeoff between security and flexibility in this new class of systems.

We have implemented the Scriptroute design and deployed it on servers across 42 PlanetLab sites. The Scriptroute code is publicly available [52] and can be used for local measurement script development or for participation in the global system. To test the system, we have used this initial deployment to run RPT, a tool we created to measure routing trees around a destination, and to repeat the experiment used to evaluate GNP [40], a recently proposed Internet distance estimation technique. We find that our system will be flexible enough to implement a variety of new measurement tools despite its security restrictions, that access to many remote vantage points makes the system valuable, and that scripting is an apt choice for expressing and combining measurement tasks.

The rest of the paper is organized as follows. We describe our goals and approach in the next section and the design of our system in Section 3. We present implementation details such as the default configuration in Section 4. We evaluate our approach using two applications as case studies in Section 5, then conclude.

Measurement Tool	Measures	Support
pathchar [25]	Hop-by-hop b/w	✓[1]
pchar [34]	Hop-by-hop b/w	✓[1]
clink [15]	Hop-by-hop b/w	✓[1]
pathrate [13]	Bottleneck b/w	✓
pathload [27]	Available b/w	✓
sprobe [49]	Bottleneck b/w	✓
nettimer [33]	Bottleneck b/w	✓
bprobe [6]	Bottleneck b/w	✓
cprobe [5]	Congestion	✓
traceroute [26]	Path and RTT	✓
tcptraceroute [56]	Path and RTT	✓
ping	Round trip time	✓
zing [45]	Poisson RTT	✓
ally [53]	Alias resolution	✓[2]
tbit [41]	End-host TCP impl.	✓[2]
king [21]	Estimated RTT	✓[2]
nmap [16]	End-host services	✓[2]
treno [37]	TCP b/w	✗[3]
wping [36]	TCP b/w	✗[3]
iperf [55]	TCP b/w, loss	✗[3]
netperf [29]	TCP b/w	✗[3]
ttcp	TCP b/w	✗[3]
sting [50]	One-way loss	✗[4]
fsd [19]	Router processing	✗[5]

[1] May require an excessively high rate of traffic to execute quickly, which would be limited by policy.
[2] Measures end host properties: supported, so may simplify development, but unnecessary.
[3] Must keep a window of packets in flight, so are not supported by our synchronous interface.
[4] Supported by design, but unimplemented: requires safe raw sockets [47] or kernel firewall support.
[5] Requires address spoofing.

Table 1: Some active measurement tools supported by the design.

2 Goals and Approach

In this section, we describe our design philosophy and the approach that follows from it.

2.1 Philosophy

Our high-level goal is to foster the deployment of a community platform for distributed Internet measurement.

To be provided by the community, this platform must allow different organizations to manage their own portions of the infrastructure. To be of broad use, this platform must see widespread deployment to provide many measurement vantage points. To be of lasting value, this platform must be capable of hosting new measurement techniques.

While these goals are straightforward, achieving them is not: many promising systems fail to achieve widespread adoption. We observe two salient characteristics in successful collaborative systems, such as Gnutella and the Web. First, they are open: all users may contribute and participate. Second, they are valuable to the participants: there is benefit to both service users and providers.

Our philosophy is derived from interpreting these qualities in our domain of network measurement. To be open, we take the position that all users must be able to obtain useful levels of service by default and with negligible prior investments. If all users are authorized to obtain the same service then, just as a public Web server, there is no need to authenticate users further than their IP address. To provide value, we observe that the most compelling use of measurement staples such as traceroute and ping is not for network research, but for operational purposes. Indeed, the array of public traceroute servers is heavily populated by ISPs providing vantage points from which they may check routing and connectivity. We thus seek to seed our system with operationally useful measurement tools.

This philosophy leads to the essential conflict in the design of our system: flexibility versus security. Flexibility is required if we are to support unforeseen measurement tools. At the same time, supporting unauthenticated users poses serious security concerns. To be deployed, Scriptroute cannot serve as a vehicle that facilitates denial-of-service attacks on third parties, nor can it expose its host to attack. We next describe our approach to flexibility, then security.

2.2 Flexible Measurement Tools

Our goal is to provide Scriptroute servers with sufficient extensibility mechanisms that they can implement unanticipated measurement tools. While we cannot prove we can handle all possible new tools, we can design a system that supports their likely space. To define the space, we first considered existing active measurement tools, a sample of which (most from [11]) is shown in Table 1.

We observe that existing tools send a wide variety of types and sequences of packets, with different timing patterns, and using different methods of data analysis. Most of the tools, including some that measure bandwidth, require only a modest level of bandwidth and processing to be useful, and they do not impose tight timing coupling between the reception of one packet and the transmission of the next. The variability in functional details and modest resource requirements of these tools lead us to an architecture where measurements are supported by shipping measurement code to Scriptroute servers. This code is then interpreted in a resource-limited sandbox that includes an API for sending and receiving measurement packets and for reporting results back to the client.

We can also observe from Table 1 that there is a class of tools that need not be supported from distributed vantage points. Tools such as tbit and nmap, for example, probe properties of the endpoint being measured. They can readily be run from any vantage point to obtain the desired measurement. Similarly, tools such as King [21] work by finding unwitting proxy nodes as vantage points. These kind of tools are not targeted as part of the design of Scriptroute; we focus on tools that measure the properties of network paths that can only be observed by using Scriptroute servers themselves as vantage points.

2.3 Protecting Scriptroute Servers

We require that Scriptroute servers not expose their host to unwanted attack, despite an architecture where measurements are scripted and servers execute them on behalf of unauthenticated (and hence untrusted) clients. There are two aspects to protecting servers: restricting access and controlling resource consumption.

To isolate measurements from the host system, servers execute measurement scripts in the strongest sandbox we can construct that provides only a very narrow interface for sending and receiving packets and communicating results to the client. The design of this resource-limited sandbox is described in Section 3.

To ensure that measurement scripts do not consume enough resources to cause denial-of-service to the host, the Scriptroute server limits all aspects of measurement execution. Servers limit the duration, traffic rate, memory footprint, processor time, and number of concurrent measurements, reclaiming their resources as scripts terminate. Limits on the duration of measurements ensure that resources are replenished for subsequent measurements. Such limits prohibit long-lived experiments, but do away with allocation and reservation machinery. Similarly, measurements are not allowed access to local

Prevention Mechanism	Attack Classes Prevented
Verify packet is well-formed	Ping of Death [31]
Verify source address	UDP packet storm using echo/chargen [8]
Deny fragments	Overlapping IP fragments with conflicting data[9]
Deny ICMP error messages	Spurious host unreachable [12]
Deny broadcast	Smurfing [10]
SYNs rate-limited	SYN flooding [7]
Rate-limit traffic	Packet flooding (e.g. flood ping)

Table 2: Attacks prevented by Scriptroute policy. The top half consists of well-known "magic" packet attacks that are prevented with filters. Flooding attacks are prevented by rate-limiting.

storage, which simplifies the system but requires that the client store all intermediate state. Taken together, these limits embody a "best-effort" service model, where the Scriptroute server executes measurements only when resources are available.

2.4 Preventing Network Attacks

We require that Scriptroute servers not facilitate denial-of-service attacks on other parties, either by acting individually or as a whole. Unfortunately, this is a tall order: most Internet hosts can be unwitting participants in a denial-of-service attack, and just one unexpected packet can be interpreted as an attack by an intrusion detection system or watchful administrator. Since new attacks are discovered in existing protocol implementations with disappointing regularity, we also cannot reliably filter out attack packets at servers (e.g., by using an IDS setup) without engaging in an arms race. Instead, we set a lower bar for Scriptroute, which is that it not increase the danger to third parties, either by amplifying or laundering attacks. Attack traffic is amplified when attackers can cause many packets (or much work) to reach the target by sending few packets (or doing little work) themselves, e.g., smurfing [10]. Attack traffic is laundered when attackers cause a third party to send a packet that is not traceable to the true attacker [44].

To understand how to prevent attacks, we first considered the different kinds of attack traffic. A sample of known network attacks is listed in Table 2. We observe that these attacks fall into two classes: those that require only a few "magic" packets, and those that overwhelm targets with a flood of traffic or otherwise tie up system resources. We tackle each class differently. We also streamline the process by which recipients of unwanted measurement traffic can have it blocked.

To mitigate the first class of attacks, we block packets frequently used for attacks and infrequently needed for measurements, e.g., IP packets with broadcast destination addresses. The complete list is given in Section 4.2. We also provide accountability by ensuring that the source address of measurement traffic is that of the server and by logging client activity. The latter is possible because the TCP connection between client and server ensures that the client IP address is genuine. Together, these measures provide an identity chain that allows measurement traffic to be traced to its origin (at least, as far as Scriptroute is concerned) for more subtle attack packets that are not blocked. We note that "magic" packet attacks could be launched from anywhere in the network, probably with less effort and the same effect as via Scriptroute. That is, a Scriptroute server does not contribute to the vulnerability of the network.

The second class of attacks requires a sustained flood of traffic to arrive at the target. Our approach here is straightforward: we rate-limit measurement traffic to an acceptable, background level. This approach works well for the majority of measurement tools, many of which send small volumes of data at low rates to avoid altering the properties that they seek to measure. However, some measurement tools, primarily bandwidth estimators such as treno and pathchar, do send a large volume of high-rate traffic. We cannot safely support them in their current form and instead are hopeful that recent work on bandwidth estimation such as pathload, nettimer, and sprobe [27, 33, 49] will lead to lower rate, less intrusive tools.

We considered and discarded other approaches, such as "packet conservation," where high rates can be used, provided that send and receive traffic is roughly balanced. Unfortunately, while unbalanced traffic indicates a problem (such as high loss or deliberate discard), balanced traffic only indicates the absence of severe network congestion. Because of the best-effort nature of Internet services, request flooding (e.g., TCP connections, DNS requests) may consume nearly all available resources. Further, determining when traffic is too far out of balance is a task that depends on protocol semantics, such as delayed acknowledgements, and so it cannot be applied in a general way.

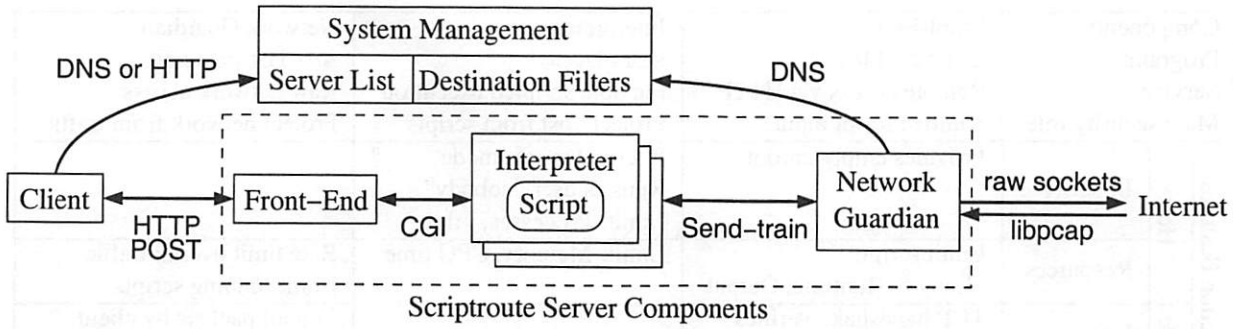


Figure 1: Scriptroute components. Clients discover servers through DNS or HTTP. Clients then submit measurements scripts to a server front-end using HTTP, which executes a new interpreter as a CGI program. Scripts running within the Interpreter use the Send-train API to send probes and receive responses. Only the network guardian can access the network, after first checking that a destination filter does not block requested probes.

Rate limits prevent a single measurement from overwhelming a destination, but we must also prevent the collection of Scriptroute servers being used for distributed denial-of-service (DDOS). Again, our approach in the short term is to rely on a sufficiently low rate limit on individual measurement that does not provide clients with leverage in terms of attack bandwidth. That is, if Scriptroute servers do not significantly amplify attack traffic levels then they do not make DDOS attacks any easier to launch.

Again, we considered more sophisticated centralized or epidemic controls that would detect groups of servers sending large volumes of traffic to the same target, e.g., by requiring that permission tokens be obtained from a pre-determined controller before starting a bandwidth-intense measurement. However, we realized that, even if the complexity issues associated with these controls can be managed, protection by destination IP address (or destination IP prefix) is not sufficient. This is because hosts other than the apparent destination can be saturated by attackers with a modest understanding of current network routes. That is, the target is not always apparent from the measurement traffic, and without a sophisticated understanding of network topology and routing, no centralized controller is in a position to prevent attackers from concentrating traffic. We expect this to be an area of further research as we gain experience.

3 System Design

In this section, we describe the components of the Scriptroute system, how these components communicate, and how a user submits a script for execution.

The set of cooperating components is shown in Figure 1. We separate these components for robustness and security: each performs a simple task, and compromising one does not help compromise others. The task of the front end Web server is to pass measurement scripts uploaded from clients to the interpreter for execution. The interpreter runs in a restricted environment and may fail by exceeding resource limits or by measurement script error. The interpreter's use of the Scriptroute API is carried by a local TCP socket to the network guardian. By separating the interpreter into its own process, it can fail without affecting the network guardian or front end. The network guardian is the only component that needs to be run with special permissions to read and write raw packets.

Each component also has a role in providing security, summarized in Figure 2. The front-end verifies that scripts are submitted from unforged IP addresses (via TCP handshaking) and prevents scripts from running too long or sending too much output. The interpreter provides flexibility in choosing what sort of probe packets to send and when, but restricts execution to a resource-limited sandbox. The practice of combining a sandbox based on a safe language with a narrow interface is well established [2, 18, 23]. Finally, the network guardian enforces rate limits and packet filtering policy, and only permits responses to probes to be returned to the measurement script. The local administrator controls the resource limits and filtering policy.

We now describe the design of each of these components in the order visited by an executing measurement.

3.1 System Management

Scriptroute servers publish their existence in a dynamically updated DNS database. This allows clients to find Scriptroute servers using descriptive host names,

Component:			Front-End	Interpreter	Network Guardian
Program:			`thttpd` [48]	`srrubycgi`	`scriptrouted`
Service:			Remote access via HTTP	Flexible scripted execution	Raw network access
Main security role:			Sanitize script input	Protect host from scripts	Protect network from traffic
Protection Features	Host	Integrity	Provides empty chroot	Interpreter safe mode [b] Runs as user "nobody" Limit: Processes, Files [c]	
		Resources	Limit script: [a] Length, Runtime, Output	Limit: Memory, CPU time	Rate limit overall traffic Limit running scripts
	Network	Integrity	TCP handshake verifies client IP address		Log all packets by client Filter dangerous packets
		Resources			Rate limit SYN packets Rate limit by destination

[a] `thttpd` provides these features by default, otherwise these limits would have to be enforced by the interpreter.
[b] Sandboxing scripts is not necessary when running an interpreter as a local user.
[c] Provides redundant protection to reinforce the safe mode against fork() and open().

Figure 2: Scriptroute server components, annotated with their security roles and features that provide host and network security.

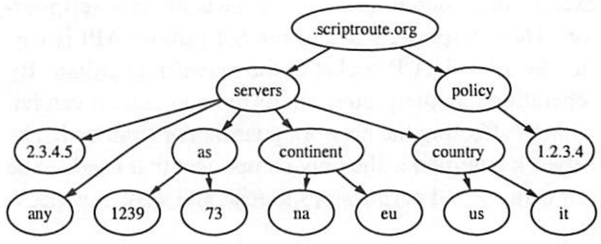


Figure 3: Scriptroute DNS name space.

and servers to publish their feature set (e.g., software version.)[1] Different Scriptroute servers may belong to different groups and use different DNS servers; ours is rooted at `scriptroute.org`. As shown in Figure 3, the name-space is separated into two subtrees: `policy` and `servers`.

The servers subtree returns pseudo-random lists of Scriptroute servers, optionally chosen by AS, country, or continent. This breakdown was chosen for convenience, but the complete database can be accessed from a dynamically generated Web page.

The policy subtree includes entries for measurement targets that wish to block unwanted measurement traffic. The goal of this repository is to restrict traffic from compliant Scriptroute servers in a single step. There are two ways to update this database. Individual targets can connect to a Web server and block measurement traffic back to their own IP address. Alternately email from a domain administrator is used for blocking traffic to entire IP prefixes. The Web interface provides a timely update when it is clear, by the TCP handshake, that a user of the target machine has requested a filter; changes are immediately propagated into the DNS policy subtree. The email-based interface deals with many hosts in the same administrative domain, but requires human verification before coarser filters are installed or removed.

[1]In contrast, traceroute servers are found using directories maintained by hand [32] and research testbeds have a static host list.

3.2 Server Front-End

Each Scriptroute server runs an ordinary Web server on port 3355, which provides a gateway for script submission and administrative tasks. There are three main "pages" on the server: job submission, traceback, and informational.

The job submission page provides an HTTP POST interface for measurement script submission, then replies with the output of the measurement. Again, the TCP handshake demonstrates that the source IP address is valid to provide a measure of accountability. A convenient feature of thttpd [48] is that it limits the execution time, size, and output of the script. We also limit the number of concurrent requests per client (1) and the number of concurrent requests overall (10). If the interpreter fails due to resource limits, the connection is closed signaling an error to the client. Unhandled exceptions in the measurement script itself are handled by the interpreter and returned to the client as text.

The traceback page provides limited access to the logs to reduce anonymity and prevent Scriptroute from "laun-

```
#! /usr/local/bin/srinterpreter

probe = Scriptroute::Udp.new(12)
probe.ip_dst = ARGV[0]
unreach = false
puts "Traceroute to #{ARGV[0]} (#{probe.ip_dst})"

catch (:unreachable) do
  ( 1..64 ).each { |ttl|
   ( 1..3 ).each { |rep|
     probe.ip_ttl = ttl
     packets = Scriptroute::send_train([ Struct::DelayedPacket.new(0,probe) ])
     response = (packets[0].response) ?  packets[0].response.packet : nil
     if(response) then
       puts '%d %s %5.3f ms' % [ ttl, response.ip_src, (packets[0].rtt * 1000.0) ]
       if(response.is_a?(Scriptroute::Icmp)) then
         unreach = true if(response.icmp_type == Scriptroute::ICMP_UNREACH)
       end
     else
       puts ttl.to_s + ' *'
     end
     $stdout.flush
   }
   throw :unreachable if(unreach)
  }
end
```
Figure 4: Traceroute, as implemented in Ruby for Scriptroute. For comparison, a stripped down version of traceroute [14] is implemented in 200 lines of C.

dering" traffic. Specifically, it provides the tcpdump-formatted packets sent to particular IP addresses along with the address of the corresponding client.

Finally, the informational page provides information about the measurement traffic supported, how to contact the administrator of the server, how to learn more about Scriptroute, and how to add destination filters to block unwanted measurement traffic. So that administrators know where to look to when their systems receive unexpected measurement traffic, we encourage Scriptroute servers that also have a port 80 Web server to link this page, to direct concerns to the central management site.

3.3 Script Interpreter

The front end pipes submitted jobs to a scripting language interpreter in a new process. In our implementation, we chose Ruby, but any language that supports a strong sandbox can be used. The interpreter runs as a separate process so that it can fail independently: aggressive kernel resource limits are used to prevent significant resource consumption; when exceeded, the process terminates abruptly.

The interpreter provides access to the Scriptroute API and a simplified interface to packet contents, taking care of such details as network byte ordering. The measurement script can instantiate new packets, fill them in, then send them via the Send-train API call, which the interpreter translates into a socket connection to the network guardian. An example script implementing traceroute is shown in Figure 4.

The interpreter communicates to the network guardian using only the Send-train API. Send-train supports most network measurements by sending a train of probe packets and collecting their responses. The Send-train operation takes an array of (delay, probe packet) pairs as an argument, then returns an array of (time-stamp, probe packet, time-stamp, response packet) tuples. The observation is that most measurements send a train of probes (possibly just one) then wait for the responses and repeat.

3.4 Network Guardian

The network guardian is responsible for limiting the rate of measurement traffic and regulating the type of packets sent. It combines destination-specific filters to block traffic as stored in DNS with the rate limits and additional filters configured by the local administrator.

To support the Send-train API, the network guardian is responsible for matching probes with their responses, which protects the host from measurement tools that might otherwise see unrelated traffic. Matching responses to probes is simple in the case of traceroute-like

UDP probes and ICMP error responses (which match the encapsulated header), ICMP echo request/response (which match the sequence number), and unsolicited TCP probes with TCP RST responses (which match the address, port, and sequence number). It is more complex for TCP connections, where we match responses to the earliest plausible probe.[2]

The network guardian mediates access to the raw sockets and packet capture facilities of the kernel, so must be run "as root" or with special configuration. Finally, the network guardian logs sent and received packets with the client that requested the corresponding measurement. These logs can be used after the fact to infer what sort of traffic might have offended a remote site. We describe the policies enforced by the network guardian in detail in the next section.

4 Implementation

In this section, we describe the implementation of the interpreter and network guardian. We describe the default policy configuration that protects the network and destination hosts. The network guardian consists of 3,000 lines of C, and the interpreter adds another 600, calling on Ruby and tcpdump as libraries.

The system management interface is a combination of a Web server (thttpd), a DNS server (tinydns), and a small daemon that updates the zone file based on registration messages sent by servers and destination filters submitted by Web and email. Implementation details of this component are straightforward and not described further.

4.1 Script Interpreter

The interpreter provides an environment to support measurement scripts and hand packet trains to the network guardian. It creates a sandbox with a name space that includes the Scriptroute API and class definitions for standard packet types.

The class-based packet interface simplifies development by attending to details such as network byte ordering and host name lookup. The packet class's `to_s` (to string) method uses code from tcpdump to present a familiar representation of the packet for debugging.

[2]To establish a TCP connection in a measurement requires that the host kernel not see the SYN/ACK and respond immediately with a RST. This can be prevented using safe raw sockets [47] or the kernel's firewall [50].

The interpreter uses the kernel to limit the script's resource consumption in processor time (4 second default) and memory footprint (50K stack, 50K data, 8MB address space, though these limits depend on the operating system). Each of these limits is configurable by the local administrator. Additional resource limits on concurrently opened file handles (7) and processes (1) are used to reinforce the interpreter's safe mode against inadvertent calls to `open()` or `fork()`. Scripts that exceed these limits are abruptly terminated, which is why each script executes in its own interpreter process.

Resource limits on individual processes must be combined with a limit on the number of concurrent measurement scripts. A new interpreter requests permission to execute from the network guardian, and may be told to try again later if there are too many scripts in the system or too many scripts being executed on behalf of the same user (the default limits are one per user to a maximum of ten per system). A user is defined by the client IP address if accessed through the front end, or by the user name of the process if executed locally.

The *chroot* environment created by the front-end is inherited by the interpreter. A chroot-ed process executes with all file accesses confined to sub-tree of the file system. While not designed for sandboxing processes, it can be used to isolate processes from from the rest of the machine, in this case preventing the interpreter from accessing any files in the system. We make the chroot robust to common attacks by both running the interpreter as "nobody," which lacks permission to write the filesystem, and keeping the chroot empty; it contains only the statically-linked interpreter and the sent packet logfile.

We chose Ruby because it is a lightweight, type-safe, general-purpose interpreted language with a safe mode that guards access to system calls. While most of these features are just convenient, a flexible safe mode is essential. For example, Ruby's safe mode prevents files and sockets from being opened, but permits the script to write its results back to the client over an already existing socket. We believed that a scripting language would make development simple, which was an important consideration given that many existing tools would need to be ported. We believed that choosing a general-purpose language was important for encouraging adoption: those who already know Ruby should find it trivial to write measurement scripts, and those who are new to the language can apply their new experience to ordinary tasks. Finally, we found that the Ruby interpreter integrated well with C, which was important because the isolation enforced by the safe mode prevents the script from accessing the network guardian directly.

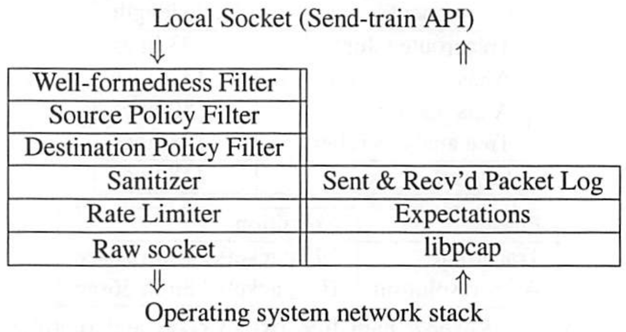


Figure 5: Architecture of the network guardian. At left is the downward path probes take out to the network; right is the upward path packets take back. On Planetlab, the raw socket and libpcap interfaces are replaced by safe raw sockets [47].

4.2 Network Guardian

The network guardian is responsible for protecting the network and destination hosts by applying policy checks before traffic is sent. It is the only component that requires special privilege to read and write a raw socket. It provides this packet generation service using the Send-train API to interpreters (or any other process) on the local machine. The architecture of the network guardian is shown in Figure 5. We describe the components in the order they are visited by a measurement.

The network guardian accepts TCP socket connections on the localhost address from the interpreter. Listening only to localhost allows the network guardian to operate on behalf of local processes without providing remote service, adding a small measure of security. The interface across this socket is text-based for extensibility and ease of debugging. However, binary packets must be encoded to be transferred across a text-based interface; we chose base64 encoding, a method commonly used for encoding MIME attachments.

Packets face a series of verification steps. First, they are checked for integrity and that the reflector can recognize likely responses to the probe. For example, this verifies the packet has sufficient length for its headers and is of a known protocol.

Second, the source's filter is applied. The administrator of the Scriptroute server has discretion over what traffic should be generated, and can decide what packets can be sent. The default source filters remove broadcast and multicast packets, IP fragments, ICMP error messages, TCP resets, UDP and TCP traffic to "priviledged" ports (those below 1024) other than 80 (HTTP) and 53 (DNS), and traffic to the local host and subnet.

Bucket	Recharge rate	Burst size
Measurement SYN	1 packet/s	4 packets
Destination I	1 Kbytes/s	8 Kbytes
Destination II	3 packet/s	10 packets
Source	3 Kbytes/s	100 Kbytes

Table 3: Default rate limit parameters: each can be adjusted by the local administrator. The source and destination rate limits are shared by all measurements, while the SYN limit applies to each measurement.

Third, the destination policy is applied. The network guardian executes a lookup on the destination address in the policy subtree of the DNS described in Section 3.1. A filter may be stored (as a TXT record and in BPF [38] format) under the destination's address. If no entry exists for that destination, no additional filters are applied. The filter is cached for five minutes, but if the DNS server is unreachable, the previously cached entry is used.

Fourth, packets are "sanitized" by setting the source address to that of the local machine and setting the source port, if UDP or TCP, to one owned by the network guardian. This prevents harmful interactions with other traffic on the same machine and provides accountability by avoiding source spoofing. The packet is then checksummed.

As a final step, the probes are scheduled to be sent by passing them through a series of rate-limiting token buckets. The default burst size (bucket depth) and recharge rate parameters of these buckets are shown in Table 3. If the packet is a TCP SYN, it is passed through a per-experiment rate-limiter that is intended to prevent SYN flooding attacks. Next, the packet passes through per-destination limiting to prevent flooding attacks. The first per-destination limit is on the rate of traffic in bytes to prevent bandwidth-consuming flooding attacks. The second limit is on the rate of packets sent, because a packet represents some overhead at the destination, possibly involving application-layer processing. The final rate limiter prevents excessive bandwidth consumption at the source.

When probes are sent, "expectation" state is created, representing the set of possible responses to associate with the probe. For example, sending an ICMP echo request creates the expectation of either an ICMP echo response or an ICMP error message. These expectations filter the packets read from libpcap – preventing unrelated traffic from escaping to measurement scripts – and match responses with probes, simplifying tool development.

Matched probes and responses with their timestamps, or sent probes that received no response after a timeout period, constitute the response to the Send-train API. The reflector logs each probe/response pair before returning it to the interpreter, ending with the status message "done."

5 Evaluation

Applications are the key to evaluating Scriptroute. That is, the most important evaluation questions are: what new measurements does Scriptroute enable, how readily can they be expressed, and how efficiently are they run? To begin to answer these questions, we used Scriptroute for two case studies. First, we use Scriptroute to implement a new debugging tool, "reverse path tree" (RPT), that gathers and summarizes network routes towards a target. Second, we use Scriptroute to gather a dataset suitable for assessing the merits of Global Network Positioning (GNP), a newly proposed Internet distance prediction technique. Both of these case studies were undertaken primarily for the purpose of evaluating the capabilities of Scriptroute. At the same time, both represent real tasks that could not be accomplished without access to many measurement vantage points.

5.1 Reverse Path Tree (RPT)

By "reverse path tree" we mean the tree of routes that are used to reach a specific host from other locations on the Internet, as opposed to paths from that host outwards to other locations that are provided by regular traceroute. The reverse path tree summarizes how a host is reached from the rest of the Internet, and it can only be generated with the help of remote hosts. It generalizes the practice of ISPs manually using a remote traceroute server to check connectivity and routing to themselves.

The Scriptroute-based RPT discovery tool proceeds in two logical steps: tracing the routes from as many servers as possible to the destination; and merging them with IP alias resolution to recognize interface IP addresses that belong to the same router [20, 53]. Scriptroute provides the opportunity to reduce the amount of traffic needed to construct the tree by recognizing segments that have already been traversed on-line. In contrast, assembling a tree from standard traceroutes would probe routers close to the destination repeatedly. We do this by embedding a list of previously observed IP addresses in the measurement script, having the script terminate when it reaches a part of the tree that has already been mapped, and mapping from different servers

Component	Code length
Traceroute (shipped)	33 lines
Alias res. (interpreted)	137 lines
Alias res. (client)	50 lines
Tree analysis (client)	148 lines
Overall	318 lines

Phase	Execution
Traceroute	195 packets / 1min 20sec
Alias resolution	102 packets / 3min 36sec

Table 4: Reverse path tree (RPT) code and runtime statistics. Components are "shipped" to remote Scriptroute servers, "interpreted" by the local Scriptroute server, or executed as part of the "client's" analysis

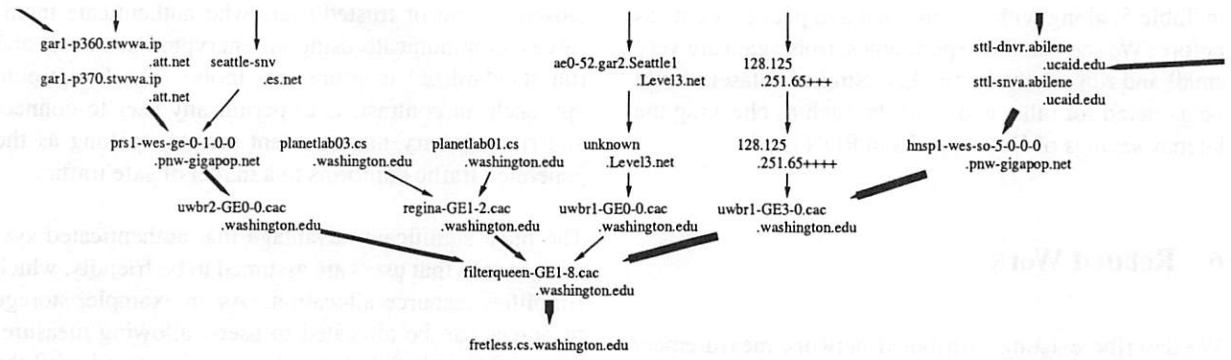
sequentially. This reduction allows the system to scale without loading the network. We also note that alias resolution is run on the local Scriptroute daemon. It does not need to be distributed because it measures endpoint rather than path properties.

A sample tree mapped by RPT to one of our hosts is shown in Figures 6 and 7. Already we can see that Scriptroute deployment on PlanetLab provides a rich enough set of servers to construct a useful tree. Code size and runtime statistics for the RPT tool are given in Table 4. Code size shows the number of lines of code at the client and shipped to Scriptroute servers. The number of packets includes only measurement traffic, and the execution time for each phase is given. These phases could be overlapped, but performance is already adequate for the task. We can see that both client and server code is small; the choice of a scripting language for constructing tools appears worthwhile. Further, Scriptroute supports a useful measurement task despite the rate limits imposed for security.

We expect RPT to serve as a foundation for future tools to infer the location of performance problems by observing which parts of the tree are shared between Scriptroute servers. For example, Scriptroute could be used to measure loss between each server and a destination, as well as trace the tree. Techniques such as [35, 42] could then be used to pinpoint lossy segments.

5.2 Validating GNP

To demonstrate the value of Scriptroute for network measurement research, we undertook to validate claims for Global Network Positioning (GNP) [40], a recently proposed technique for estimating Internet latency between points. GNP estimates latency using multi-dimensional mappings derived from measurements between each point and special landmarks. The details



garl-p360.stwwa.ip .att.net
garl-p370.stwwa.ip .att.net
seattle-snv .es.net
prs1-wes-ge-0-1-0-0 .pnw-gigapop.net
planetlab03.cs .washington.edu
planetlab01.cs .washington.edu
uwbr2-GE0-0.cac .washington.edu
regina-GE1-2.cac .washington.edu
ae0-52.gar2.Seattle1 .Level3.net
unknown .Level3.net
uwbr1-GE0-0.cac .washington.edu
128.125 .251.65+++
128.125 .251.65++++
uwbr1-GE3-0.cac .washington.edu
sttl-dnvr.abilene .ucaid.edu
sttl-snva.abilene .ucaid.edu
hnsp1-wes-so-5-0-0-0 .pnw-gigapop.net
filterqueen-GE1-8.cac .washington.edu
fretless.cs.washington.edu

Figure 6: An excerpt from a reverse path tree measured by Scriptroute. Line thickness represents the number of paths that traverse the link. Aliases are listed together. Most of the tree is to the right and above: this is the neighborhood of the root. Those IP addresses listed with '+' are unresponsive successors of an IP address.

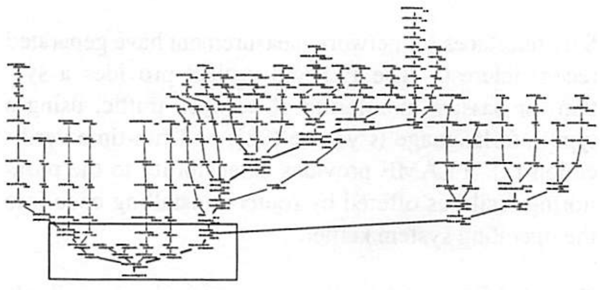


Figure 7: The complete reverse path tree, of which Figure 6 is an excerpt. This overall view shows the detail available by combining many vantage points.

of GNP itself are unimportant for this paper; our aim is simply to demonstrate the utility and scalability of Scriptroute by repeating a real experiment. For this analysis, we require a dataset consisting of measured latencies between Scriptroute servers and many Internet hosts. These measurements can then be compared against GNP-derived estimates. The GNP study required the authors to obtain accounts on 19 machines distributed around the globe – we would like to make this sort of measurement study nearly trivial.

As input to the GNP analysis tool, we gathered a set of latency measurements from 31 Scriptroute servers as vantage points and roughly 3200 other Internet hosts from a previously selected database. This is actually a considerably larger dataset than that used in [40]. Each Scriptroute script pinged a random selection of ten hosts at a time and returned the minimum round trip latency to the client. Each host was pinged 15 times, rather than 220 as in [40].

We used these latency measurements as a dataset to evaluate the accuracy of GNP estimates. Fifteen Scriptroute servers are designated as landmarks, and we use the GNP analysis tool to compare GNP estimates to measured latencies between the non-landmark Scriptroute servers and the hosts. In Figure 8, we plot both our results for the cumulative distribution of relative error (as defined in [40]) and the results from the data set used in [40]. We find a slightly higher relative error, but on the whole the results are comparable, despite our lack of tuning.

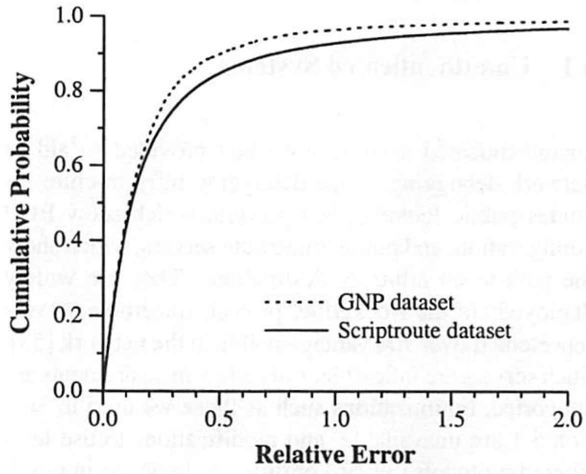


Figure 8: Relative error distribution of GNP.

This experiment highlights the capabilities of Scriptroute as a tool for gathering network performance data. The code size for the client and server scripts is given

Component	Code Length
Ping sweep (shipped)	33 lines
Analysis (client)	55 lines
Ping packets	60K per server 1.8M overall
Execution time	2 hours 30 minutes (in parallel)

Table 5: GNP dataset collector statistics.

in Table 5, along with the run time and packet counts as before. We see that the experiment scripts again are very small and run relatively quickly. Similar datasets could be gathered for other experiments, such as checking the latency savings of Detour paths in RON [3, 51].

6 Related Work

We describe existing distributed network measurement and debugging systems classified by whether they support unauthenticated clients, as this is a key feature of our design. We then describe safe local interfaces for network measurement that share attributes of the Scriptroute software architecture.

6.1 Unauthenticated Systems

Unauthenticated systems are often provided to aid in network debugging. Such debugging infrastructure includes public looking-glass servers, which show BGP configuration, and public traceroute servers, which show the path to an arbitrary destination. They are widely deployed: in the Rocketfuel project, traceroute servers represented over 700 vantage points in the network [53]. Such servers are inflexible: only a few measurements are supported, optimizations such as those we used in Section 5.1 are unavailable, and modifications to use less-filtered protocols [39, 56] or different logic are impossible. These servers are often tedious to use cooperatively: they may come and go faster than Web directories can be updated, and often use distinct interfaces. Scriptroute was designed to address these problems while building on the successes of these unauthenticated systems.

6.2 Authenticated Systems

Systems for network research, including Netbed (formerly emulab) [57], NIMI [45], Surveyor [30], IPMA [24], AMP [1], and RON [3]. From our perspective, these systems are similar, so we describe the most established one, NIMI. The National Internet Measurement Infrastructure (NIMI) is a research platform for distributed network measurement. Their design focus was on scalability and security, and a goal of their project was to support standardized network metrics from the IETF's IPPM working group [46].

NIMI, and the Network Probe Daemon upon which it is based, have similar goals as Scriptroute but different approaches. The NIMI approach to security is one of a closed system of trusted users who authenticate themselves, communicate using an encrypted protocol, and run standardized measurement tools. The Scriptroute approach, in contrast, is to permit any user to connect and run arbitrary measurement scripts, so long as the generated traffic conforms to a model of safe traffic.

The most significant advantage that authenticated systems have is that users are assumed to be friendly, which simplifies resource allocation. As an example, storage resources can be allocated to users, allowing measurements to be scheduled and their results stored until the user returns to claim them.

6.3 Extensible Network Measurement

Safe interfaces for network measurement have generated recent interest. The FLAME project provides a system for passive monitoring of network traffic, using a type-safe language (Cyclone [28]) and run-time verification [2]. FLAME provides extensibility to the monitoring facilities offered by routers, installing code into the operating system kernel.

Two projects support active measurements on a single system using a similar API. The PeriScope [22] project provides a kernel API to send groups of ICMP echo requests without returning to user space, which they argue helps accuracy. Pásztor and Veitch [43] also separate measurement logic from sending probe packets in different processes, but they do so for precisely scheduled packet transmission using a real time task in RTLinux. Scriptroute complements these systems by providing a layer between scripts and the kernel that can be extended to support these richer interfaces. Scriptroute currently supports raw sockets with libpcap by default, and Scout's safe raw sockets on Planetlab, allowing measurement scripts to transparently take advantage of new host operating system features.

7 Conclusions and Future Work

We have presented the design and implementation of Scriptroute, a new platform that allows ordinary Internet users to make network measurements from remote vantage points. Scriptroute is motivated by the popularity and utility of public traceroute servers. Clients locate servers using the DNS and ship measurement tasks as scripts. This provides the flexibility to implement a variety of non-intrusive tools for measuring path properties and makes it easy to coordinate measurements across servers. To protect servers from abuse, measurement scripts are executed in a resource-limited sandbox controlled by the local administrator. To prevent the system from being used to launch denial-of-service attacks, measurement traffic is checked, rate-limited, and logged for accountability.

The Scriptroute software is publicly available [52], including clients and sample measurement scripts, as well as the server and interpreter source. We have deployed servers across the PlanetLab testbed of 42 sites. We have used the resulting system to measure routing trees around a destination and to collect a latency dataset suitable for evaluating Internet distance prediction techniques. Our early experience suggests that the system is quite flexible and useful, despite its security restrictions, and that scripting is an apt choice for expressing and combining measurement tasks.

We view Scriptroute as a work in progress. We believe that Scriptroute shows how a public infrastructure can substantially improve our ability to make safe, flexible network measurements. With experience, we hope to improve the system and better assess our design choices. Some interesting features are not yet implemented, including support for measurements using TCP connections and tools that send responses rather than probes. We also expect our security policies to evolve as we uncover patterns of preferred usage and attempted abuse, and as our model of safe network measurement traffic is broadened with the advent of new tools.

Acknowledgements

We wish to thank Intel Research for providing access to their PlanetLab resources. Mike Wawrzoniak and Andy Bavier made it possible for Scriptroute to run using PlanetLab's safe raw sockets. Vern Paxson provided helpful comments. We also thank David Richardson, Brent Chun, and Ratul Mahajan.

This work was supported by DARPA under grant no. F30602-00-2-0565.

References

[1] Active Measurement Project. `http://amp.nlanr.net/`.

[2] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, and M. Greenwald. Open packet monitoring on FLAME: Safety, performance and applications. In *IFIP Int'l Working Conference on Active Networks (IWAN)*, 2002.

[3] D. Anderson, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, 2002.

[4] J. Bellardo and S. Savage. Measuring packet reordering. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[5] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. Tech. Rep. TR-96-006, Boston University CS Dept., 1996.

[6] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. In *IEEE INFOCOM*, 1997.

[7] CERT. TCP SYN flooding and IP spoofing attacks. `http://www.cert.org/advisories/CA-1996-21.html`, 1996.

[8] CERT. UDP port denial of service attack. `http://www.cert.org/advisories/CA-1996-01.html`, 1996.

[9] CERT. IP denial of service attacks. `http://www.cert.org/advisories/CA-1997-28.html`, 1997.

[10] CERT. Smurf IP denial of service attacks. `http://www.cert.org/advisories/CA-1998-01.html`, 1998.

[11] Cooperative Association for Internet Data Analysis (CAIDA). Internet tools taxonomy. `http://www.caida.org/tools/taxonomy/`, 2002.

[12] Cowzilla and P. Dreamer. Puke. `http://www.cotse.com/sw/dos/icmp/puke.c`, 1996.

[13] C. Dovrolis, P. Ramanathan, and D. Moore. What do packet dispersion techniques measure? In *IEEE INFOCOM*, 2001.

[14] A. B. Downey. trout. `http://rocky.wellesley.edu/downey/trout/`, 1999.

[15] A. B. Downey. Using pathchar to estimate Internet link characteristics. In *ACM SIGCOMM*, 1999.

[16] Fyodor. NMAP: The network mapper. `http://www.insecure.org/nmap/`.

[17] E. Gavron. NANOG traceroute. `ftp://ftp.login.com/pub/software/traceroute/beta/`.

[18] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *USENIX Security Symposium*, 1996.

[19] R. Govindan and V. Paxson. Estimating router ICMP generation delays. In *Passive & Active Measurement (PAM)*, 2002.

[20] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *IEEE INFOCOM*, 2000.

[21] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[22] K. Harfoush, A. Bestavros, and J. Byers. PeriScope: An active measurement API. In *Passive & Active Measurement (PAM)*, 2002.

[23] C. Hawblitzel, *et al.* Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, 1998.

[24] Internet Performance Measurement and Analysis (IPMA) project. `http://www.merit.edu/ipma/`, 2002.

[25] V. Jacobson. Pathchar. `ftp://ftp.ee.lbl.gov/pathchar`.

[26] V. Jacobson. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.Z`.

[27] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In *ACM SIGCOMM*, 2002.

[28] T. Jim, *et al.* Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[29] R. Jones. Netperf. `http://www.netperf.org/`.

[30] S. Kalidindi and M. J. Zekauskas. Surveyor: An infrastructure for Internet performance measurements. In *INET'99*, 1999.

[31] M. Kenney, ed. Ping o' death. `http://www.insecure.org/sploits/ping-o-death.html`, 1996.

[32] T. Kernen. traceroute.org. `http://www.traceroute.org`.

[33] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *USITS*, 2001.

[34] B. Mah. Estimating bandwidth and other network properties. In *Internet Statistics and Metrics Analysis Workshop*, 2000.

[35] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[36] M. Mathis. Windowed ping: an IP layer performance diagnostic. In *INET'94/JENC5*, 1994.

[37] M. Mathis. Diagnosing Internet congestion with a transport layer performance tool. In *INET'96*, 1996.

[38] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter Technical Conference*, 1993.

[39] N. McCarthy. fft. `http://www.mainnerve.com/fft/`.

[40] T. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.

[41] J. Padhye and S. Floyd. Identifying the TCP behavior of Web servers. In *ACM SIGCOMM*, 2001.

[42] V. N. Padmanabhan, L. Qiu, and H. J. Wang. Passive network tomography using bayesian inference. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[43] A. Pásztor and D. Veitch. A precision infrastructure for active probing. In *Passive & Active Measurement (PAM)*, 2001.

[44] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM Computer Communication Review*, 2001.

[45] V. Paxson, A. Adams, and M. Mathis. Experiences with NIMI. In *Passive & Active Measurement (PAM)*, 2000.

[46] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. RFC 2330, 1998.

[47] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *HotNets-I*, 2002.

[48] J. Poskanzer. thttpd. `http://www.acme.com/software/thttpd/`.

[49] S. Saroiu, P. K. Gummadi, and S. D. Gribble. Sprobe: A fast technique for measuring bottleneck bandwidth in uncooperative environments. In *Submitted for publication*, 2002. `http://sprobe.cs.washington.edu/sprobe.ps`.

[50] S. Savage. Sting: a TCP-based network measurement tool. In *USITS*, 1999.

[51] S. Savage, *et al.* The end-to-end effects of Internet path selection. In *ACM SIGCOMM*, 1999.

[52] Scriptroute. `http://www.scriptroute.org/`.

[53] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *ACM SIGCOMM*, 2002.

[54] H. Tangmunarunkit, R. Govindan, D. Estrin, and S. Shenker. The impact of routing policy on Internet paths. In *IEEE INFOCOM*, 2001.

[55] A. Tirumala, F. Qin, J. Dugan, and J. Ferguson. Iperf. `http://dast.nlanr.net/Projects/Iperf/`, 2002.

[56] M. C. Toren. tcptraceroute. `http://michael.toren.net/code/tcptraceroute/`.

[57] B. White, *et al.* An integrated experimental environment for distributed systems and network. In *OSDI*, 2002.

Network-Sensitive Service Discovery

An-Cheng Huang and Peter Steenkiste
Carnegie Mellon University
{pach,prs}@cs.cmu.edu

Abstract

We consider the problem of network-sensitive service selection (NSSS): finding services that match a particular set of functional and network properties. Current solutions handle this problem using a two-step process. First, a user obtains a list of candidates through service discovery. Then, the user applies a network-sensitive server selection technique to find the best service. Such approaches are complex and expensive since each user has to solve the NSSS problem independently. In this paper, we present a simple alternative: network-sensitive service discovery (NSSD). By integrating network-sensitivity into the service discovery process, NSSD allows users who are looking for services to specify both the desired functional and network properties at the same time. Users benefit since they only have to solve a restricted version of the server selection problem. Moreover, NSSD can solve the NSSS problem more efficiently by amortizing the overhead over many users. We present the design of NSSD, a prototype implementation, and experimental results that illustrate how NSSD can be utilized for different applications.

1 Introduction

There is considerable interest in network services that are more sophisticated than traditional Web services. Examples include interactive services such as video conferencing, distributed games, and distance learning. An important property of these services is that their performance depends critically on the network, so the service selection process should consider the network capabilities, i.e., it should be *network-sensitive*. One way of building such sophisticated services is through composition of service components (e.g., automatic path creation [23] in Ninja [13], Panda [31], and Libra [34]). The "brokers" that select and compose the service components also have to consider network properties, e.g., they should make sure sufficient bandwidth is available between a component and the users, and between components. The key operation in these examples is the network-sensitive service selection (NSSS) problem.

Existing research related to the NSSS problem falls in roughly two classes. *Service discovery* infrastructures allow a user to find a set of servers with certain functional properties, where the properties are typically described as a set of attribute-value pairs. Existing solutions, for example, [16] and [6], differ in the flexibility of naming and matching and in their scalability properties. On the other hand, given a set of servers that provide a particular service, the *server selection* problem is to select the one that can best satisfy the user's requirements. Previous work in this area has mostly focused on the problem of identifying the Web server that is "closest" to a client, e.g., has the highest-bandwidth or lowest-latency path [3, 4, 21, 32].

Clearly we can solve the NSSS problem by combining service discovery and server selection in the following way. A user first utilizes a service discovery infrastructure to obtain a list of servers that can provide the required service and then applies a server selection technique to identify the server that can best satisfy the user's requirements. Unfortunately, this solution is often not practical. A first problem is scalability: for common services, the number of servers to be considered in the server selection phase would be overwhelming, making the process very expensive. A second problem is complexity: server selection is a difficult problem, and requiring each user (e.g., application or broker) to develop these mechanisms from scratch increases the development cost. Finally, if services are offered by commercial providers, they may not be willing to expose their individual servers, so user-side selection cannot be performed.

We propose a simple alternative, namely *network-sensitive service discovery* (NSSD). The idea is that when users contact the service discovery infrastructure, they can specify not only static functional properties, but also network connectivity and server load properties. NSSD returns the best server (or a small number of good servers) according to the user's requirements. Therefore, in effect we have moved most of the complexity of server selection from the user to the service discovery infrastructure. Providers do not need to expose all their servers to users, since server selection is handled by NSSD, which is trusted by the providers.

Moreover, by integrating service discovery and server selection, NSSD can amortize overhead such as collecting network information across multiple users and can also apply distributed solutions. Therefore, we can solve the NSSS problem more efficiently and in a more scalable way.

The remainder of this paper is organized as follows. We elaborate on challenges associated with the NSSS problem in the next section. In Section 3 we describe our network-sensitive service discovery solution, including the design of the service API and the mechanisms used for network-sensitive service discovery. We describe a prototype system in Section 4, and Section 5 presents an evaluation using the PlanetLab testbed. Finally, we discuss related work and summarize.

2 Problem Statement

2.1 Service Model

Let us first identify the players:

- *Providers*: Each provider provides a *service*, for example, a service that provides streaming video of movies. Typically, a provider will use a set of distributed *servers* to deliver the service, for example, a set of replicated streaming servers.
- *Users*: Each user invokes and uses one or more services. A user can either be an *end user*, for example, a person watching a movie using a streaming video service, or another service provider, for example, a video streaming service may rely on another service provider for transcoding service.

For a particular service type, there will typically be many different providers, each of which may have many servers. In a traditional setup, a user utilizes a service discovery mechanism to select a provider that can deliver the required service. The selection of the specific server can then either be done by the selected provider (*provider-side selection*), or it can be done by the user, assuming the provider makes information about its internal capabilities available to the user (*user-side selection*).

Unfortunately, server selection becomes more complicated if the selection has to be done in a network-sensitive fashion or if users want to use user-specific selection criteria. Before we address these questions, let us look at some application examples.

2.2 Applications

In Figure 1, four users want to play a multiplayer online game together, so they need a game server to host their gaming session. Specifically, they need a game server that not only satisfies certain functional properties (e.g., supports a particular game and has certain anti-cheating features) but also delivers good "performance". For this

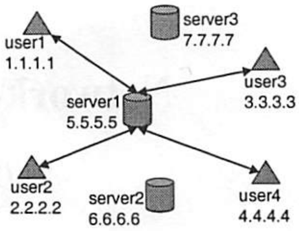


Figure 1: Selecting a game server for a multiplayer online game

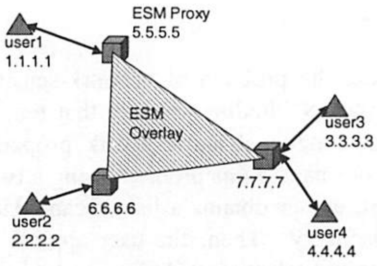


Figure 2: Proxy-based End System Multicast (ESM) example

application, this means that the players have a low network latency to the server and that the server is lightly loaded.

In the second example (Figure 2), a group of users want to use a proxy-based End System Multicast (ESM, see [5]) service. Each participant sends its packets to an ESM proxy, and all proxies serving this group construct an ESM overlay to efficiently deliver the packets to all other participants. The problem is finding a set of appropriate ESM proxies to serve the users. One possible selection metric in this case is the sum of the latencies between each user and the ESM proxy assigned to the user, since minimizing this metric may reduce the total network resource usage.

The third example is a simple service composition scenario (Figure 3): a user wants a service that provides low bit rate MPEG-4 video streams, and we can deliver such a service by putting together a high bit rate MPEG-2 video streaming service and an MPEG-2-to-MPEG-4 video transcoding service. If our goal is to minimize the total bandwidth usage, this scenario presents an interesting situation: selecting a streaming server close to the user can reduce bandwidth usage, but a more distant streaming server may turn out to be a better choice if we can find a transcoder that is very close to the server. In other words, this problem requires the coordinated selection of multiple services.

2.3 Current Solutions

There has been a lot of research on network-sensitive server selection, i.e., given a list of servers, select the best one considering certain network properties. Some approaches perform active probing at selection time to

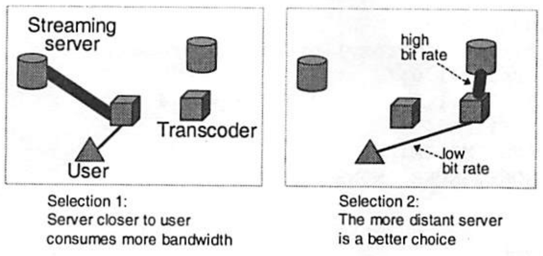

Streaming server
Transcoder
User
Selection 1:
Server closer to user consumes more bandwidth
high bit rate
low bit rate
Selection 2:
The more distant server is a better choice

Figure 3: Composing a video streaming service

determine the best server, for example, [4] and [8], while others use network properties that have been computed, measured, or observed earlier, for example, [3, 30, 32, 33]. Most of these techniques were developed for specific applications, and they were applied either on the *user side* or on the *provider side*.

In a provider-side approach, a user first utilizes service discovery to select a provider, and the provider then internally applies a network-sensitive selection technique to select one of its servers for the user. In a user-side approach, a user obtains a list of candidate servers from the selected provider, and it then applies network-sensitive selection to select one of the candidates.

Although applying network sensitivity on the user side or the provider side is suitable for some applications (e.g., user-side Web mirror server selection, provider-side cache selection in content distribution networks, etc.), both user-side and provider-side approaches have their limitations. The advantage of provider-side approaches, i.e., being transparent to users, is also its disadvantage: a user does not control the selection criteria and does not know what level of performance can be expected. User-side approaches, on the other hand, have high overhead: every user needs to obtain a potentially long list of candidates and collect network status information for the candidates. Also, each application has to implement a network-sensitive selection technique. Moreover, providers must release details about their internal capabilities, a dubious assumption at best.

In this paper, we propose a different approach: *network-sensitive service discovery* (NSSD), which integrates service discovery and network-sensitive server selection. NSSD allows a user to specify both functional and network properties when selecting a server. Given these properties, NSSD returns the best server or a small number of "good" candidates to the user. The user can then use the candidates to perform user-specific optimizations. There may be competing NSSD infrastructures, and providers and users can choose which one to use. A large provider providing many servers/services can even implement its own NSSD infrastructure.

Compared with provider-side approaches, NSSD allows users to control the selection criteria, and it exposes sufficient information to allow users to further optimize the selections (e.g., global optimizations in service composition). Compared with user-side approaches, NSSD has lower overhead both in terms of service discovery (only a small number of good candidates are returned to the user) and network sensitivity (NSSD can amortize the cost of collecting network status information). In addition, providers only need to release their complete server information to a "trusted" NSSD infrastructure.

3 Network-Sensitive Service Discovery

We define the API used for formulating NSSD queries and describe several possible NSSD designs.

3.1 NSSD API

To define our API for NSSD, we need to determine what functionalities NSSD should provide. From the three examples in Section 2.2, we can see that the server selection problem is an optimization problem: find a solution (e.g., a game server) that optimizes a certain metric (e.g., the maximum latency) for a target or a set of targets (e.g., the players). The examples illustrate two types of optimization problems: *local* and *global*. A local optimization problem involves the selection of a single service, e.g., select a game server that minimizes the maximum latency to a group of players. A global optimization problem involves the *coordinated* selection of different services, e.g., select a streaming server and a transcoder such that the overall bandwidth usage is minimized. Note that in service composition scenarios (e.g., Figure 3), selecting each service independently using local optimizations may not yield the globally optimal solution.

The key question is: what part of the optimization should be done by NSSD and what part should be left to the user. We decided to define an API that would allow a user to ask NSSD to perform local optimizations for individual services using generic optimization metrics. However, a user can also ask NSSD to return a small number of "locally good" candidates for each service so that the user can then apply global optimization criteria across services using the returned results.

Let us elaborate on these design decisions:

- *Local optimization only*: Our API allows users to specify local optimization problems only, i.e., a user can only ask NSSD to select one service at a time. The reason for this decision is that global optimizations are likely to be service-specific and may be arbitrarily complex, so it is more appropriate to leave them to the user. Note, however, that multiple "identical servers" may be required to provide a service. For example, in Figure 2, we need to select three ESM proxies for the ESM service, using the sum of the latencies as the optimization metric. We consider such optimization problems to be local, and they are

```
FindServers(
  service_properties, // service attributes
  target_list,        // optimization targets
  num_servers,        // num. of identical servers needed
  num_solutions,      // num. of solutions needed
  latency_type,       // MAX/AVG/NONE
  latency_constraint, // constraint/MINIMIZE/NONE
  bw_type,            // MIN/AVG/NONE
  bw_constraint,      // constraint/MAXIMIZE/NONE
  load_constraint     // constraint/MINIMIZE/NONE
         )

Output
  solution            // solution(s)
  mapping             // user-server mapping(s)
  fitness             // the ''score(s)'' of the solution(s)
```

Figure 4: The NSSD API

supported by NSSD.

- *A set of standard metrics*: Our API allows a user to specify constraints and preferences on a set of standard metrics: maximum and average latency, minimum and average bandwidth, and server load. While it may be possible to allow users to specify, for example, their own utility functions based on these standard metrics, it is currently not clear that the potential benefit is worth the extra complexity.
- *Best-n-solutions*: As defined by the first two decisions, NSSD supports local optimizations based on a set of standard metrics. While this is sufficient in many cases involving the selection of single services, it is not adequate in cases requiring the coordinated selection of multiple services or the use of user-specific metrics. Therefore, NSSD also allows a user to ask for a number of good solutions when selecting a service. The user can then apply user-specific or global optimization criteria on these locally good solutions. In Section 3.2, we will use a service composition example to illustrate the use of this best-n-solutions feature.

Figure 4 shows the resulting NSSD API. The argument "num_servers" specifies how many identical servers are required in a solution, and "num_solutions" specifies how many locally good solutions should be returned. The solutions are returned in "solution", in the form of IP addresses. When multiple identical servers are needed in a solution, "mapping" specifies which server is used for each user. NSSD also returns a "fitness" (i.e., the value of the selection metric) for each returned solution, which may be helpful to the user for use in further optimizations. This can be viewed as a compromise between two extremes: having the provider not release any details about the service it can provide (which prevents user-specific optimizations) and having the provider list all its capabilities (which would be needed for pure user-side optimizations). Note that a server returned may be

```
FindServers(
  "(type=ESMProxy)(protocol=Narada)\
(version=1.0)",
  "1.1.1.1,2.2.2.2,3.3.3.3,4.4.4.4",
  3, 1,
  AVG, MINIMIZE,
  NONE, NONE, NONE
         )

Output
  solution:      {"5.5.5.5,6.6.6.6,7.7.7.7"}
  mapping:       {"0,1,2,2"}
  fitness:       {42.42}
```

Figure 5: Using the API in the ESM example

the front end of a cluster of servers. Since servers in a cluster have similar network properties, this does not affect the effectiveness of the network-sensitive selection in NSSD, and it leaves room for the provider to do internal load balancing. Note also that additional information regarding each solution (e.g., monetary cost) may need to be returned so that the user can perform further optimizations.

Let us use the End System Multicast (ESM) example in Figure 2 to illustrate the use of the API. The top part of Figure 5 shows the NSSD query in this scenario. We first specify that we want to find ESM proxies that are using the Narada protocol version 1.0. The next parameter specify that the selection should be optimized for the four users in this scenario. The next two parameters specify that we want three identical ESM proxies in a solution, and we only need the best solution. The remaining input parameters specify that we want to minimize the average latency, and we do not have constraints or preferences on bandwidth and load. Assuming that the best solution is the configuration in Figure 2, the result returned by the API is shown in the bottom part of Figure 5. NSSD returns the best solution, which consists of three ESM proxies, and the "mapping" specifies that user 1 is assigned to ESM proxy 5.5.5.5, user 2 is assigned to 6.6.6.6, and users 3 and 4 are assigned to 7.7.7.7. In addition, the fitness value of this solution is 42.42.

3.2 A Sample Application: Service Composition

In this section, we use a more complex example of service composition to illustrate the use of NSSD. Suppose the five users in Figure 6 want to establish a video conferencing session. The problem is that they are using different video conferencing applications and hardware platforms: P1 and P2 are using the MBone applications vic/SDR, P3 and P4 are using Microsoft NetMeeting, and P5 is using a handheld device that is receive-only and does not do protocol negotiation. We can support this conference by composing a video conferencing service using the following more primitive service components. First, we need a "video conferencing gateway"

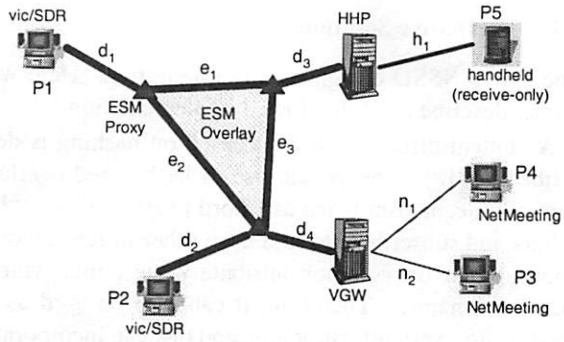


Figure 6: An example of service composition: video conferencing session

(VGW) that supports interoperability between H.323, which is used by NetMeeting, and Session Initiation Protocol (SIP), which is used by SDR. Second, we need a "handheld proxy" (HHP) that can join the session on behalf of the handheld user and forward the video stream to the handheld device. Finally, we can use the ESM service to establish a multicast session among the vic users, the VGW, and the HHP, as shown in Figure 6. Note that this is more general than service composition based on a *path model*, which is explored in, for example, the Ninja [23] and Panda [31] projects.

This service composition example raises an interesting challenge: the selection of the different components is *mutually dependent*. For example, suppose our goal is to minimize the total network resource usage, and the optimization heuristic we use is to minimize the following function:

$$\begin{aligned} &W_1(d_1+d_2+d_3+d_4+\frac{2}{3}(e_1+e_2+e_3)) \\ &\quad +W_2h_1 \\ &\quad +W_3(n_1+n_2) \end{aligned} \tag{3.1}$$

where W_1, W_2, and W_3 are weights for the three parts (multicast, handheld, and NetMeeting) of the service, and the other variables are the latencies between different nodes as depicted in Figure 6. Different weights for the three parts reflect the difference in bandwidth consumption. For example, NetMeeting can only receive one video stream, and the HHP may reduce the video bandwidth before forwarding it to the handheld.

Unfortunately, selecting all the components together to minimize the above function may be too expensive to be practical. For example, suppose there are n VGWs, n HHPs, and n ESM proxies available. To find the optimal configuration, we need to look at roughly n^5 possible configurations, which is only feasible when n is small. Therefore, our approach is to use the following heuristic: select each component using local optimizations and then combine the locally optimal solutions into a global solution. This is the approach supported by NSSD. For example, in the video conferencing service above, we first ask NSSD to return the VGW that is closest to the NetMeeting users (since they only support unicast), we then ask NSSD to return the HHP that is closest to the handheld user, and finally we ask NSSD to return the optimal set of ESM proxies (minimizing $d_1+d_2+d_3+d_4$) for the vic/SDR users, the selected VGW, and the selected HHP. In other words, we utilize NSSD to solve three local optimization problems sequentially (using three local optimization heuristics) and combine the three local solutions to get a global solution.

Of course, a combination of locally optimal solutions may not be globally optimal. To improve the component selection, we can utilize the "best-n-solutions" functionality provided by NSSD. For example, in the video conferencing service, we first ask NSSD to return the best n VGWs (in terms of latency to NetMeeting users), and we then ask NSSD to return the best (closest to the handheld user) m HHPs. Now we have nm possible VGW/HHP combinations. For each of the nm combinations, we ask NSSD to find the optimal set of ESM proxies, and we get a global solution by combining the VGW, the HHP, and the ESM proxies. Therefore, we have a set of nm global solutions, and we can use Function 3.1 to evaluate them and select the best global solution in this set.

We believe this approach can yield a reasonably good global solution and allow a provider to adjust the tradeoff between global optimality and optimization cost by controlling the values of n and m. For example, if n and m are set to 1, the resulting solution is simply a combination of locally optimal solutions. On the other hand, if n and m are the total numbers of VGWs and HHPs, respectively, we are in fact performing an exhaustive search in the complete search space, and we will find the globally optimal solution at a higher cost. Later in Section 5.4 we will use this video conferencing example to evaluate the effectiveness of this approach.

3.3 A Simple NSSD Query Processor

Given the NSSD API described in Section 3.1, NSSD queries can be resolved in many different ways. In this section, we first describe a simple approach that heavily leverages earlier work in service discovery and network measurement; this is also the design used in our prototype. We mention alternative approaches in the next section.

As shown in Figure 7, a simple NSSD query processor (QP) can be built on top of a service discovery infrastructure (e.g., the Service Location Protocol [16]) and a network measurement infrastructure that can estimate the network properties (e.g., latency) between nodes. When the QP module receives an NSSD query (step 1 in Figure 7), it forwards the functional part of the query to the

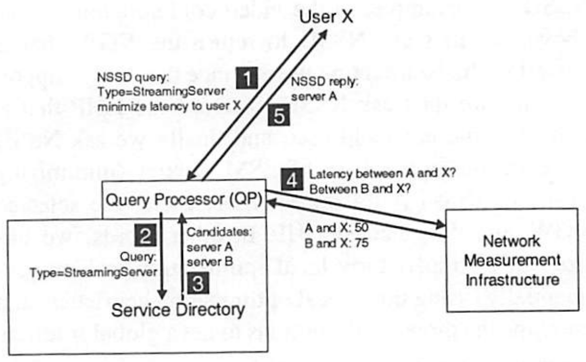


Figure 7: Handling an NSSD query

service directory (step 2). The directory returns a list of candidates that match the functional properties specified in the query (step 3). Then the QP module retrieves the necessary network information (e.g., the latency between each of the candidate and user X) from the network measurement infrastructure (step 4). Finally, the QP module computes the best solution as described below and returns it to the user (step 5).

One benefit of integrating service discovery and server selection in a single service is that caching can be used to improve performance and scalability. When NSSD gets requests from many users, the cost of collecting network status information or server load information can be amortized. Furthermore, network nodes that are close to each other should have similar network properties. Therefore, nodes can be aggregated to reduce the amount of information required (for example, use the same latency for all users in an address prefix). This should improve the effectiveness of caching.

Selecting the best solution(s) requires solving a fairly general optimization problem. Through the NSSD API described earlier, a user can specify many different combinations of (1) constraints and preferences on three metrics (latency, bandwidth, and load), (2) how many identical servers are needed in a solution, and (3) how many solutions are needed. The QP computes the solution for a query as follows. First, the QP applies any constraints in a query to eliminate any ineligible candidates. Then, the preferences in a query can be formulated as an optimization problem. If there is only a single preference, the QP can simply sort the candidates accordingly. If there are multiple preferences (e.g., minimize load and minimize latency), there may not be any candidates that satisfy all preferences. One possible solution is to have the QP define an order among the preferences (e.g., in the order they appear in the query) and sort the candidates accordingly. Finally, if a query requests multiple identical servers in a solution (e.g., requesting 3 ESM proxies for 4 users), the optimization problem can be cast as *p*-median, *p*-center, or set covering problems [7], which are more expensive to solve.

3.4 Alternative Solutions

The above NSSD design is only one option. Here we briefly describe three distributed implementations.

A content discovery system based on hashing is described in [12]. The system uses a hash-based overlay network mechanism (such as Chord [35]) to allow publishers and subscribers to find each other in rendezvous points based on common attribute-value pairs, which may be dynamic. Therefore, it can also be used as a service discovery infrastructure, and one can incorporate network sensitivity into the query resolution phase of the system so that the returned matches satisfy certain network properties specified in the query.

Another alternative is application-layer anycasting [38], in which each service is represented by an anycast domain name (ADN). A user submits an ADN along with a server selection filter (which specifies the selection criteria) to an anycast resolver, which resolves the ADN to a list of IP addresses and selects one or more from the list using the filter. Potentially, the ADN and resolvers can be extended to allow users to specify the desired service attributes, and the filter can be generalized to support more general metrics.

Finally, distributed routing algorithms are highly scalable, and they can, for example, be used to find a path that satisfies certain network properties and also includes a server with certain available computational resources [19]. A generalization of this approach can be combined with a service discovery mechanism to handle NSSD queries.

4 Implementation

We describe a prototype NSSD based on Service Location Protocol (SLP) [16] and Global Network Positioning (GNP) [24]. We have experimented with versions of our NSSD implementation on the ABone [1], Emulab [9], and PlanetLab [29] testbeds.

4.1 Extending SLP

Our implementation of NSSD is based on OpenSLP [26], an implementation of SLP. Available services register their information (location, type, and attributes) with a Directory Agent (DA), and users looking for services send queries (specifying the type and attributes of the desired services) to the DA. Service types and attributes are well known so that a user knows what to ask for. SLP query specification is quite general: attributes can be specified as an LDAPv3 search filter [18], which supports, for example, logical operations, inequality, and substring match. Therefore, a user query includes a service type (e.g., "GameServer") and a filter, e.g., "(&(game=Half-Life)(mod=Counter-

```
(|(&(game=Half-Life)(mod=Counter-Strike)
    (version>=1.5)(load<=10))
  (&(x-NSSD-targets=1.01,2.02;3.03,4.04)
    (x-NSSD-maxlatency=minimize)))
```

Figure 8: A sample filter for the game example

Strike)(version>=1.5))". We believe this query representation is sufficiently general to support NSSD queries as defined by the API in Section 3.1.

In order to support NSSD queries, we extended the semantics of the SLP filter to include a set of special attributes, representing the parameters described in Figure 4. For example, suppose a user wants to find a game server that (1) matches certain service attributes, (2) is serving at most ten sessions, and (3) minimizes the maximum latency for the two players whose GNP coordinates are "1.01,2.02" and "3.03,4.04", respectively. These parameters can be specified by the filter shown in Figure 8.

The original SLP API returns a list of "service URLs" [15]. To return additional information about each selected server, we simply add the information to the end of the URLs. For example, to return the mapping, the DA can return the following service URL: "service:GameServer://192.168.0.1;x-NSSD-mapping=0,0,0,0".

Since server load is also a service attribute, each server's registration includes the load attribute (e.g., "load=0.5"). When conducting "live" experiments (i.e., involving applications running on actual network hosts), we need mechanisms to dynamically update the load value of each server. Our live experiments are developed and conducted on the PlanetLab wide-area testbed [29], which allows us shell access on about 70 hosts at nearly 30 sites. We implemented a "push-based" load update mechanism: servers push their load information (in the form of a registration update) to the DA. In our evaluation, we look at how the frequency of load update affects the performance of the server selection techniques.

4.2 Network Measurement

A network measurement infrastructure provides a way for users to obtain network information. A number of such infrastructures have been proposed, for example, IDMaps [11] and Global Network Positioning (GNP) [24]. Most of these infrastructures only provide latency information. Since it is in general much harder to estimate the bandwidth between two network nodes, we currently focus on dealing with the latency metric.

In our implementation, we use GNP as the network measurement infrastructure to provide latency (round trip time) information between two network nodes. The key idea behind GNP is to model the Internet as a geometric space using a set of "landmark nodes", and each network host can compute its own coordinates by probing the landmarks. It is then straightforward to compute the distance (latency) between two hosts given their coordinates. Since the PlanetLab testbed is our current platform, we use the GNP approach to obtain a set of coordinates for every PlanetLab node.

In the GNP model, each node computes its own coordinates. Therefore, in our implementation, we use GNP coordinates as a service attribute, i.e., when a server register with the SLP DA, the registration includes the coordinates of the server node. When a user specifies the list of optimization targets in a query (see the API in Section 3.1), each target is specified in the form of GNP coordinates. When a QP asks for a list of candidates, the DA returns the list along with the coordinates of each candidate. The advantage of this design is that since the coordinates are computed off-line, and the latency information can be derived from the coordinates directly, the cost of querying the network measurement infrastructure at runtime is eliminated. A QP can simply use the candidates' and the targets' coordinates to solve the particular optimization problem specified by a user.

4.3 Selection Techniques

The API defined in Section 3.1 can be used to specify a wide range of selection techniques. Below we list the selection techniques that are currently implemented in our QP and used in our evaluation (k denotes the number of identical servers needed in a solution, and n denotes the number of locally good solutions needed).

- "$k=1, n=1$, minimize load" (**MLR**): find the server with the lowest load. If there are multiple servers with the same load, select one randomly.
- "$k=1$, $n=1$, minimize maximum latency" (**MM**): find the server that minimizes the maximum latency to the specified target(s).
- "$k=1$, $n=1$, load constraint x, minimize maximum latency" (**LCxMM**): among the servers that satisfy the specified load constraint (load $\leq x$), find the one that minimizes the maximum latency to the specified target(s).
- "$k=1$, $n=1$, minimize load, minimize maximum latency" (**MLMM**): find the server with the lowest load, and use maximum latency as the tiebreaker.
- "$k=p$, $n=1$, minimize average latency" (**PMA**): find a set of p servers and assign each target to a server so that the average latency between each target and its assigned server is minimized (i.e., the p-median problem [7]).
- "$k=1$, $n=m$, minimize average latency" (**MMA**): find the best m servers in terms of the average latency between server and each user.
- "Random" (**R**): randomly select a server.

In the evaluation section, we use these techniques to solve the NSSS problem for different applications.

5 Evaluation

We present the results of experiments that quantify how well our NSSD prototype can support two sample applications. First, we use a multiplayer gaming application to illustrate the importance of network-sensitive server selection and to show the relative performance of the different selection mechanisms in our prototype. Next we use the service composition example of Section 3.2 to show the trade-offs between local and global optimizations. We also present the computational overhead of the NSSD prototype.

The gaming service experiments are conducted on the PlanetLab wide-area testbed [29]. The service composition experiments are simulations based on latency measurement data from the Active Measurement Project at NLANR [25].

5.1 Importance of Network Sensitivity

In the first set of experiments, we look at how network sensitivity can help improve application performance. We implemented a simple "simulated" multiplayer game client/server: a number of game clients join a session hosted by a game server, and every 100ms each client sends a small UDP packet to the hosting server, which processes the packet, forwards it to all other participants in the session, and sends an "ACK" packet back to the sender. Our performance metric for the gaming service is the maximum latency from the server to all clients since we do not want sessions in which some players are very close to the server while others experience high latency. Note that the latency metric is measured from the time that a packet is sent by a client to the time that the ACK from the server is received by the client.

The goal of this set of experiments is to evaluate the role of network performance in the server selection process. Therefore, we minimized the computation overhead in the game server to ensure that the server computation power, including the number of servers, does not affect game performance. We compare four different scenarios. First, we consider two scenarios, each of which has 10 distributed servers selected from PlanetLab nodes. The **MM** and **R** techniques are applied in the two scenarios, respectively. For comparison, we also consider two centralized cluster scenarios, **CAM** and **CMU**, each of which has a 3-server cluster, and a random server is selected for each session. In the CAM scenario, the cluster is located at the University of Cambridge; in the CMU scenario, the cluster is at CMU. The user machines are randomly selected from 50 PlanetLab nodes.

Figure 9 shows the average maximum latency (of 100 sessions) as a function of the number of participants in each session. MM consistently has the lowest maximum latency, confirming that network-sensitive service discovery can successfully come up with the best solution. More specifically, the results illustrate that being able to choose from a set of distributed servers (MM outperforms CMU and CAM) in a network-sensitive fashion (MM outperforms R) is a win. Interestingly, having distributed servers is not necessarily better than a centralized solution. In our case, a centrally-located cluster (CMU) outperforms a distributed network-"insensitive" solution (R).

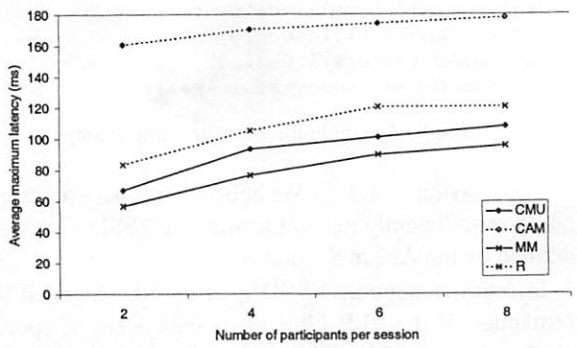


Figure 9: Central cluster vs. distributed servers in a multi-player game application

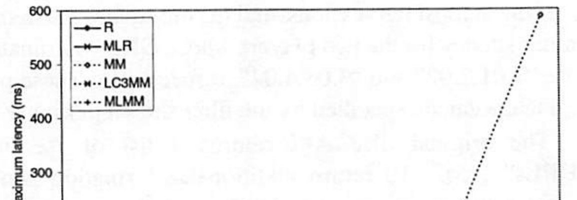


Figure 10: Effect of per-packet processing: average maximum latency

5.2 Different Selection Techniques

While the focus of this paper is not on new selection techniques, we next show how NSSD can easily support different techniques. We use the simulated game applications to compare the effectiveness of different selection techniques in NSSD. We look at the techniques **MLR, MM, LC3MM, MLMM,** and **R** described in Section 4.3. (LC3MM has load constraint 3, which means that only servers serving no more than three sessions are eligible.) Ten nodes at 10 different sites are used as game servers, 50 other nodes are game clients, and there are four randomly selected participants in each ses-

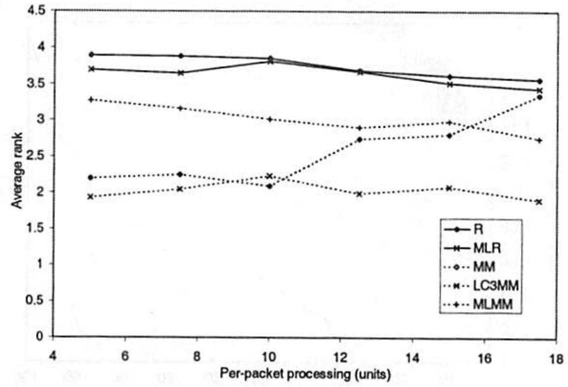


Figure 11: Effect of per-packet processing: average rank

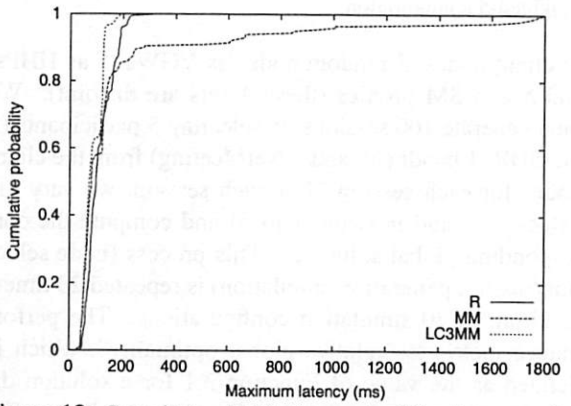


Figure 12: Cumulative distribution under 15 units per-packet processing

sion. There are on average 10 simultaneous sessions at any time, and the server load information in the DA's database is updated immediately, i.e., each server's load value is updated whenever it changes (we relax this in the next section). We ran measurements for different per-packet processing overhead, which we controlled by changing the number of iterations of a computationally expensive loop. The unit of the processing overhead corresponds roughly to 1% of the CPU (when hosting a 4-participant session). For example, on a server that is hosting a single 4-participant session with 10 units of per-packet processing overhead, the CPU load is about 10%.

Figure 10 shows the average maximum latency (of 200 sessions) as a function of the per-packet processing cost. R and MLR have almost identical performance, which is expected since in this case MLR evenly distributes sessions based on the number of gaming sessions on each server. MM and LC3MM are also almost identical when the per-packet processing is low. However, the performance of MM degrades rapidly at higher loads, while LC3MM remains the best performer throughout this set of experiments.

Figure 11 shows the results from a different perspective: it shows the average "rank" of the five techniques,

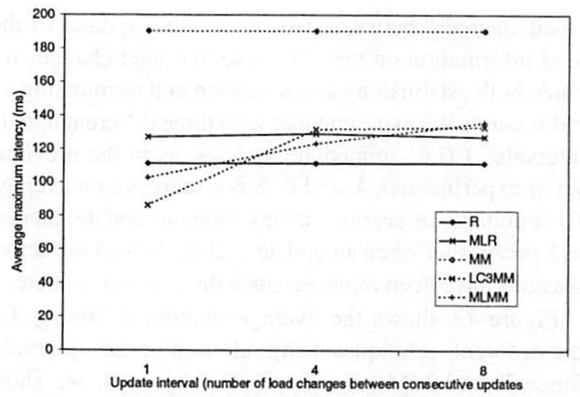


Figure 13: Effect of load update frequency: average maximum latency

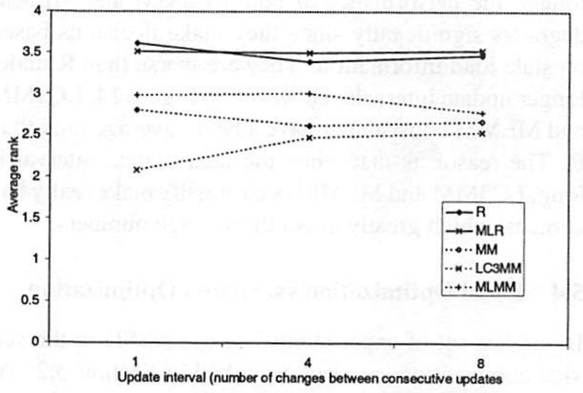


Figure 14: Effect of load update frequency: average rank

i.e., for each session, we rank the techniques from 1 to 5 (best to worst), and we then average over the 200 sessions. We see that although the rank of MM gets worse for higher loads, it is still better than R and MLR at the highest load, despite the fact that its average is much worse than those of R and MLR. The reason can be seen in Figure 12, which compares the cumulative distributions of the maximum latency of R, MM, and LC3MM for the case of 15 units per-packet processing. It shows that MM in fact makes many good selections, which helps its average rank. However, the 10% really bad choices (selecting overloaded servers) hurt the average significantly. In contrast, LC3MM consistently makes good selections, which makes it the best performer both in terms of average and rank.

5.3 Effect of Load Update Frequency

In the previous set of experiments, the server load information stored in the DA's database is always up-to-date, which is not feasible in practice. We now look at how the load update frequency affects the performance of the selection techniques. The experimental set up is the same as that in the previous section, except that we fix the per-packet processing to 15 units and vary the load update interval, which is defined as the number of system-wide "load changes" between two consecutive updates of the load information on the DA. Note that load changes include both establishing a new session and terminating an old session. We experimented with three different update intervals: 1 (i.e., immediate updates, as in the previous set of experiments), 4, and 8. Since there are on average 10 simultaneous sessions at any time, an update interval of 8 means that when an update occurs, 40% of the active sessions have been replaced since the previous update.

Figure 13 shows the average maximum latency for the different techniques using different update intervals. Since R and MM are not affected by load, we show the data from the previous set of experiments for comparison. We see that as the update interval becomes longer, the performance of both LC3MM and MLMM degrades significantly since they make decisions based on stale load information. They are worse than R under longer update intervals. However, in Figure 14, LC3MM and MLMM consistently have a better average rank than R. The reason is that when the load update interval is long, LC3MM and MLMM occasionally make really bad choices, which greatly affect the average numbers.

5.4 Local Optimization vs. Global Optimization

In the last set of experiments, we use NSSD in the service composition scenario described in Section 3.2. As depicted in Figure 6, we need to find a video conferencing gateway (VGW), a handheld proxy (HHP), and a set of ESM proxies. We use the heuristic described in Section 3.2: first, we ask NSSD to select the best n VGWs using the MMA technique (see Section 4.3), and then we ask NSSD to select the best m HHPs using MMA. For each of the nm VGW/HHP combinations, we ask NSSD to find the optimal set of ESM proxies using the PMA technique. Finally, we evaluate the resulting nm global solutions using Function 3.1 and select the best one.

Since we are only interested in the global optimality (as defined by Function 3.1) of the resulting service instances, we do not need to run the various components on actual machines. Therefore, our experiments are based on simulations using the latency measurement data from the NLANR Active Measurement Project [25]. The data consists of round trip time measurements from each of 130 monitors (mostly located in the U.S.) to all the other monitors. We process the data to generate a latency matrix for 103 sites, and then in the simulations we randomly select nodes from these sites to represent users and various components. Next, we present the results from three sets of experiments and compare them.

5.4.1 Weighted-5

In the weighted-5 set, we use weights 5.0, 2.0, and 1.0 for W_1, W_2, and W_3, respectively. We select 40 random nodes

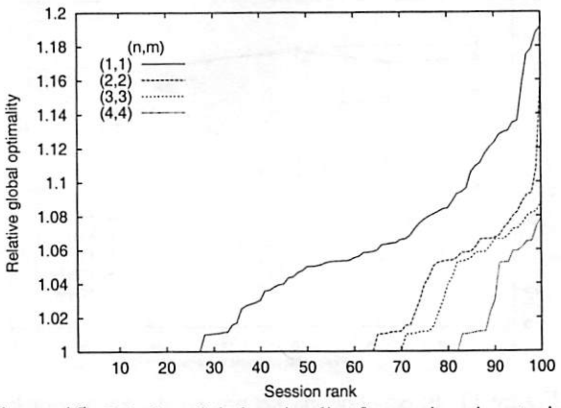


Figure 15: Relative global optimality for sessions in a typical weighted-5 configuration

as client nodes, 5 random nodes as VGWs, 5 as HHPs, and 5 as ESM proxies (these 4 sets are disjoint). We then generate 100 sessions by selecting 5 participants (2 vic/SDR, 1 handheld, and 2 NetMeeting) from the client nodes for each session. For each session, we vary the values of n and m (from 1 to 5) and compute the corresponding global solutions. This process (node selection/session generation/simulation) is repeated 20 times, resulting in 20 simulation configurations. The performance metric is "relative global optimality", which is defined as the value of Function 3.1 for a solution divided by the value for the globally optimal solution. For example, a solution with relative global optimality 1.25 is 25% worse than the globally optimal solution.

Let us first look at all 100 sessions in a typical simulation configuration. We experimented with four different settings for (n,m): (1,1), (2,2), (3,3), and (4,4), and the results from a typical configuration are plotted in Figure 15. For each (n,m) setting, the sessions are sorted according to their relative global optimality (rank 1 to 100, i.e., best to worst). When (n,m) is (1,1), we are able to find the globally optimal solution for 27 sessions, and the worst-case relative global optimality is 1.19. We can see that as we increase the size of the search space using the best-n-solutions feature of NSSD, we not only increase the chance of finding the globally optimal solution but also improve the worst-case performance. Therefore, the result demonstrates the effectiveness of our approximate approach.

Next, we want to look at the results for all sessions in all 20 simulation configurations using all (n,m) settings. The average relative global optimality result is shown in Figure 16. Each data point is the average of 20 configurations, each of which is the average of 100 sessions. We decided to present the average (mean) relative global optimality for each (n,m) setting instead of the median value because (as we can see in Figure 15) the median value often can not show the difference in the performance of different settings. When (n,m) is (1,1),

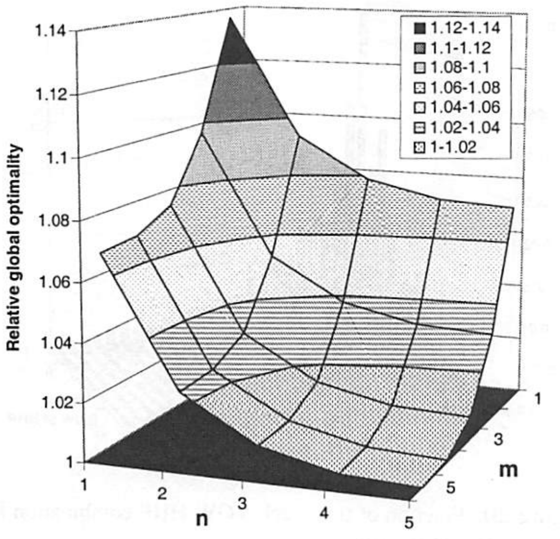


Figure 16: Relative global optimality for weighted-5

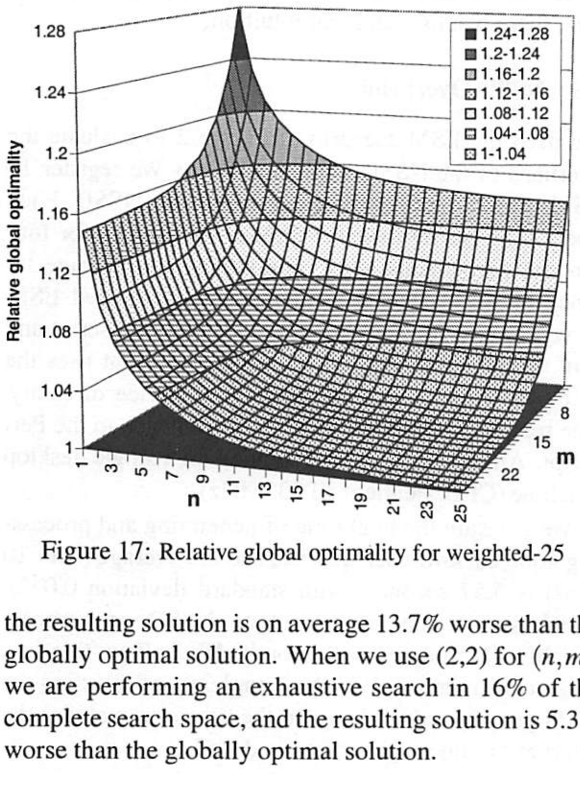


Figure 17: Relative global optimality for weighted-25

the resulting solution is on average 13.7% worse than the globally optimal solution. When we use (2,2) for (n,m), we are performing an exhaustive search in 16% of the complete search space, and the resulting solution is 5.3% worse than the globally optimal solution.

5.4.2 Weighted-25

The weighted-25 setup is the same as weighted-5 above except that we use 25 VGWs, 25 HHPs, and 25 ESM proxies for this set. Figure 17 shows the average relative global optimality for this set of experiments. When we set (n,m) to (1,1) and (10,10), the average relative global optimality of the resulting solution is 1.279 and 1.035, respectively (i.e., 27.9% and 3.5% worse than the globally optimal solution).

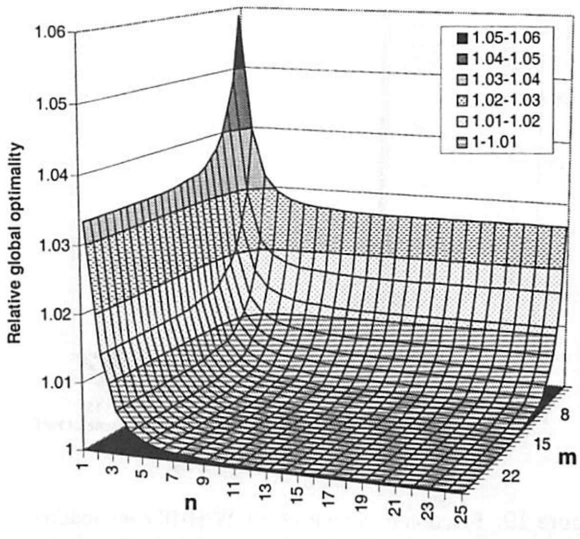


Figure 18: Relative global optimality for unweighted-25

5.4.3 Unweighted-25

The unweighted-25 set is the same as weighted-25 above except that the weights W_1, W_2, and W_3 in the global optimization function (Function 3.1) are all set to 1.0. The average relative global optimality result is shown in Figure 18. When we set (n,m) to (1,1) and (10,10), the resulting solution is on average 5.9% and 0.1% worse than the globally optimal solution, respectively.

5.4.4 Comparison

A comparison between weighted-5 and weighted-25 illustrates a few points. First, although using the combination of locally optimal solutions can greatly reduce the cost of solving the selection problem, it can lead to bad solutions (in terms of global optimality). Second, using the best-n-solutions feature of NSSD is effective, as we can significantly improve the global optimality of the resulting solution by searching in a relatively small space. Third, the performance at (1,1) seems to degrade as the complete search space becomes larger (1.137 in weighted-5 and 1.279 in weighted-25). On the other hand, the effectiveness of the best-n-solutions approach seems to increase with the size of the complete search space, e.g., when searching only 16% of the complete search space (i.e., when we set (n,m) to (2,2) and (10,10) in weighted-5 and weighted-25, respectively), the improvement in weighted-25 is greater than in weighted-5 (27.9%→3.5% vs. 13.7%→5.3%).

When comparing weighted-25 with unweighted-25, we see that in unweighted-25, the performance at (1,1) is much better than that in weighted-25 (1.059 vs. 1.279), and increasing n and m improves the global optimality much faster than it does in weighted-25, e.g., 5.9%→1.0% vs. 27.9%→12.6% when (n,m) is (4,4).

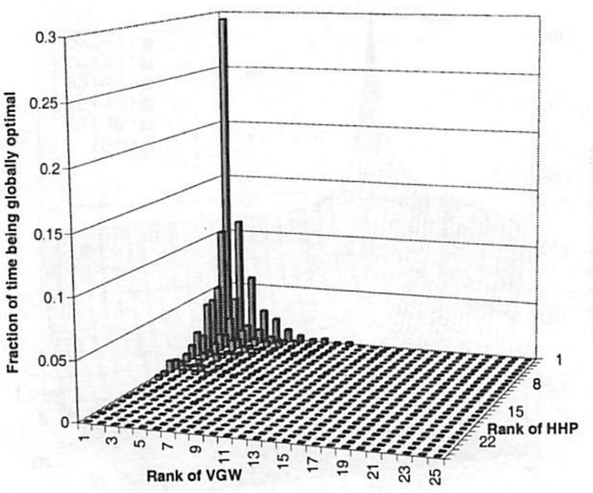


Figure 19: Fraction of time each VGW/HHP combination is globally optimal (unweighted-25)

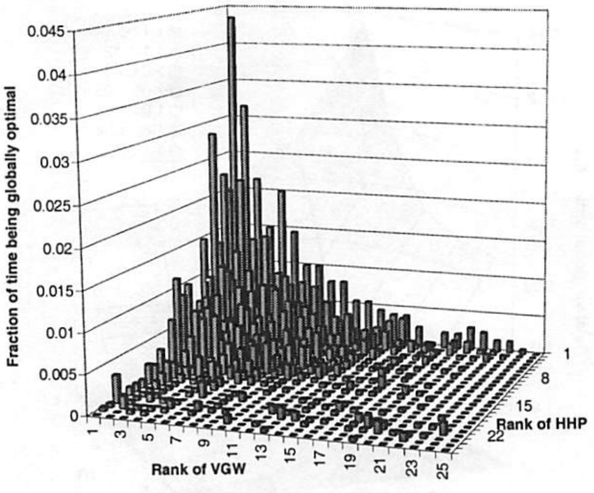


Figure 20: Fraction of time each VGW/HHP combination is globally optimal (weighted-25)

An intuitive explanation of this significant difference between the weighted and the unweighted configurations is that, in this video conferencing service composition example, the unweighted global optimization function is actually quite close to the sum of the local optimization metrics used to select the individual services. As a result, a "locally good" candidate is very likely to also be "globally good". On the other hand, in the weighted configuration, the global optimality of a solution is less dependent on the local optimality of each component, and therefore, we need to expand the search space more to find good global solutions.

To verify this explanation, we look at how likely each particular VGW/HHP combination results in the globally optimal solution. Specifically, for each of the 2000 sessions, we look at which VGW/HHP combination (according to their local ranks, e.g., the combination of the i-th ranked VGW and the j-th ranked HHP) results in the globally optimal solution. Then we aggregate the results and present, for all $1 \leq i \leq 25$ and $1 \leq j \leq 25$, the fraction of time that the combination of the i-th ranked VGW and the j-th ranked HHP results in the globally optimal solution. Figure 19 shows that in unweighted-25, nearly 30% of the time simply using the best VGW (rank 1) and the best HHP (rank 1) results in the globally optimal solution. Similarly, about 11% of the time using the 2nd-ranked VGW and the 1st-ranked HHP results in the globally optimal solution, and so on. In fact, in unweighted-25, the vast majority of globally optimal solutions involve the best few VGWs and HHPs. On the other hand, Figure 20 shows the results for weighted-25. Although using the combination of the 1st-ranked VGW and the 1st-ranked HHP is still more likely to result in the globally optimal solution than any other VGW/HHP combinations, the fraction is now only 4.3%. Furthermore, the distribution is much more dispersed. Therefore, these results match our intuition.

5.5 NSSD Overhead

We used the ESM scenario of Figure 2 to evaluate the overhead of the NSSD implementation. We register 12 ESM proxies (and 24 other services) with NSSD. Each query asks NSSD to select three ESM proxies for four random participants while minimizing the average latency between each participant and its assigned ESM proxy. The queries are generated using a Perl script and sent (through a FIFO) to another process that uses the SLP API to send the queries to the service directory. The replies flow in the reverse direction back to the Perl script. All the processes are running on a single desktop machine (CPU: Pentium III 933MHz).

We measure the total time of generating and processing 4000 back-to-back queries, and the average (over 10 runs) is 5.57 seconds (with standard deviation 0.032), which shows that in this set up NSSD can process roughly 718 queries per second. We believe this is a reasonable number given the complexity of selecting the ESM proxies and the fact that the time also includes the overhead of query generation and IPC.

6 Related Work

We summarize related work in the areas of service discovery, network measurement infrastructure, and server selection techniques.

There have been many proposals for service discovery infrastructures. For example, Service Location Protocol [16], Service Discovery Service [6], and Java-based Jini [20]. A distributed hashing-based content discovery system such as [12] can also provide service discovery functionality. These general service discovery infrastructures only support service lookup based on functional properties, not network properties. Naming-based approaches for routing client requests to appropriate servers can also provide service discovery functionality. Application-layer anycasting [38] performs server selection during anycast domain name resolution. In TRIAD [14], requests are routed according to the desired content and routing metrics. The Intentional Naming System [2] resolves intentional names, which are based on attribute-value pairs, and routes requests accordingly. Active Names [36] allows clients and service providers to customize how resolvers perform name resolution. NSSD can potentially be built on top of the service discovery and server selection mechanisms in these approaches.

An important part of NSSD is the network measurement infrastructure, which provides estimates of network properties such as latency between hosts. A number of research efforts focus on such an infrastructure. For example, in IDMaps [11], the distance between two hosts is estimated as the distance from the hosts to their nearest "tracers" plus the distance between the tracers. GNP [24] utilizes a coordinates-based approach. Remos [22] defines and implements an API for providing network information to network-aware applications.

Many network-sensitive server selection techniques have been studied before. For example, in [4] a number of probing techniques are proposed for dynamic server selection. Client clustering using BGP routing information [21] or passive monitoring [3] has been applied to server selection. Similarly, distributed binning [30] can be used to identify nodes with similar network properties. In SPAND [32], server selection is based on passive monitoring of application traffic. The effectiveness of DNS-based server selection is studied in [33]. Network-layer anycast [28] handles network-sensitive selection at the routing layer. The Smart Client architecture [37] utilizes a service-specific applet to perform server selection for a user (primarily for load-balancing and fault-transparency). The performance of various selection techniques is evaluated in [17] and [8]. These studies provide new ways of collecting and using network information for server selection and are complementary to our work. Some other efforts address the problem of request distribution in a server cluster, for example, [10] and [27]. They are also complimentary to NSSD since all nodes within the same cluster have similar network properties.

7 Conclusion

For many applications, the ability to find a server (or a set of servers) that satisfies a set of functional and network properties is very valuable. In this paper, we have proposed an integrated solution: network-sensitive service discovery (NSSD). NSSD allows users to benefit from network-sensitive selection without having to implement their own selection techniques, and it does not require providers to expose all their server information to users. The local optimization techniques supported by NSSD can be used in common cases involving the selection of individual servers, and the best-n-solutions feature provides additional information that allows users to perform user-specific global optimizations.

In our evaluation, we show that our prototype implementation of NSSD has reasonably good query processing performance. Experimental results for game server selection show that by using the local optimization functionality provided by NSSD, the simulated multiplayer game can get significant performance improvements. Simulation results for service composition demonstrate that NSSD also provides the flexibility for users to approximate the performance of global optimizations using results from local optimizations. By using the best-n-solutions feature, a user can perform global optimizations in a small search space and greatly improve the performance of the resulting solution.

Acknowledgments

This work was supported in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518. Additional support was provided by Intel. We thank the organizations hosting the PlanetLab testbed for their support. We also thank the National Laboratory for Applied Network Research for providing the latency measurement data, which was sponsored by National Science Foundation Cooperative Agreement No. ANI-9807479. We would like to thank Tze Sing Eugene Ng for his help on GNP related issues, his implementation of a heuristic search algorithm, and his comments on earlier drafts of this paper.

References

[1] Active Network Backbone (ABone). http://www.isi.edu/abone/.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of ACM SOSP '99*, Dec. 1999.

[3] M. Andrews, B. Shepherd, A. Srinivasan, P. Winkler, and F. Zane. Clustering and Server Selection using Passive Monitoring. In *Proc. IEEE INFOCOM 2002*, June 2002.

[4] R. Carter and M. Crovella. Server Selection Using Dynamic Path Characterization in Wide-Area Networks. In *Proceedings of IEEE INFOCOM '97*, Apr. 1997.

[5] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.

[6] S. E. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. Katz. An Architecture for a Secure Service Discovery Service. *MobiCOM '99*, Aug. 1999.

[7] M. S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*. John Wiley & Sons, Inc., 1995.

[8] S. G. Dykes, C. L. Jeffery, and K. A. Robbins. An Empirical Evaluation of Client-side Server Selection Algorithms. In *Proc. of IEEE INFOCOM 2000*, Mar. 2000.

[9] The Emulab testbed. http://www.emulab.net.

[10] A. Fox, S. Gribble, Y. Chawathe, and E. A. Brewer. Cluster-Based Scalable Network Services. In *Proceedings of ACM SOSP '97*, Oct. 1997.

[11] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Trans. on Networking*, 9(5):525–540, 2001.

[12] J. Gao and P. Steenkiste. Rendezvous Points-Based Scalable Content Discovery with Load Balancing. In *Proceedings of NGC 2002*, Oct. 2002.

[13] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *IEEE Computer Networks, Special Issue on Pervasive Computing*, 35(4), Mar. 2001.

[14] M. Gritter and D. R. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proceedings of USITS 2001*, Mar. 2001.

[15] E. Guttman, C. Perkins, and J. Kempf. Service Templates and Service: Schemes. RFC 2609, June 1999.

[16] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608, June 1999.

[17] J. Guyton and M. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proceedings of ACM SIGCOMM '95*, Aug. 1995.

[18] T. Howes. The String Representation of LDAP Search Filters. RFC 2254, Dec. 1997.

[19] A.-C. Huang and P. Steenkiste. Distributed Load-Sensitive Routing for Computationally-Constrained Flows. In *Proceedings of ICC 2003 (to appear)*, May 2003.

[20] Jini[tm] Network Technology. http://wwws.sun.com/software/jini/.

[21] B. Krishnamurthy and J. Wang. On Network-Aware Clustering of Web Clients. In *Proceedings of ACM SIGCOMM 2000*, Aug. 2000.

[22] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A Resource Query Interface for Network-Aware Applications. *7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.

[23] Z. M. Mao and R. H. Katz. Achieving Service Portability in ICEBERG. *IEEE GlobeCom 2000, Workshop on Service Portability (SerP-2000)*, 2000.

[24] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[25] Active Measurement Project (AMP), National Laboratory for Applied Network Research. http://watt.nlanr.net/.

[26] OpenSLP Home Page. http://www.openslp.org/.

[27] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of ASPLOS-VIII*, Oct. 1998.

[28] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. RFC 1546, Nov. 1993.

[29] PlanetLab Home Page. http://www.planet-lab.org/.

[30] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[31] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated Planning for Open Architectures. In *Proceedings for OPENARCH 2000 – Short Paper Session*, pages 17–20, Mar. 2000.

[32] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proceedings of USITS '97*, Dec. 1997.

[33] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proc. of IEEE INFOCOM 2001*, Apr. 2001.

[34] P. Steenkiste, P. Chandra, J. Gao, and U. Shah. An Active Networking Approach to Service Customization. In *Proceedings of DARPA Active Networks Conference and Exposition (DANCE'02)*, pages 305–318, May 2002.

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001*, Aug. 2001.

[36] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of USITS '99*, Oct. 1999.

[37] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of USENIX 1997 Annual Technical Conference*, Jan. 1997.

[38] E. W. Zegura, M. H. Ammar, Z. Fei, and S. Bhattacharjee. Application-Layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service. *IEEE/ACM Trans. on Networking*, 8(4):455–466, Aug. 2000.

Using Random Subsets to Build Scalable Network Services

Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat*
Department of Computer Science
Duke University
{dkostic,razor,albrecht,abhi,vahdat}@cs.duke.edu

Abstract

In this paper, we argue that a broad range of large-scale network services would benefit from a scalable mechanism for delivering state about a random subset of global participants. Key to this approach is ensuring that membership in the subset changes periodically and with uniform representation over all participants. Random subsets could help overcome inherent scaling limitations to services that maintain global state and perform global network probing. It could further improve the routing performance of peer-to-peer distributed hash tables by locating topologically-close nodes. This paper presents the design, implementation, and evaluation of *RanSub*, a scalable protocol for delivering such state.

As a first demonstration of the RanSub utility, we construct SARO, a scalable and adaptive application-layer overlay tree. SARO uses RanSub state information to locate appropriate peers for meeting application-specific delay and bandwidth targets and to dynamically adapt to changing network conditions. A large-scale evaluation of 1000 overlay nodes participating in an emulated 20,000-node wide-area network topology demonstrate both the adaptivity and scalability (in terms of per-node state and network overhead) of both RanSub and SARO. Finally, we use an existing streaming media server to distribute content through SARO running on top of the PlanetLab Internet testbed.

1 Introduction

Many distributed services must track the characteristics of a subset of their peers. This information is used for failure detection, routing, application-layer multicast, resource discovery, or update propagation. Ideally, the size of this subset would equal the number of all global participants to provide each node with the highest quality information. Unfortunately, this approach breaks down beyond a few tens of nodes across the wide-area, encountering scalability limitations both in terms of per-node state and network overhead. Recent work suggests building scalable distributed systems on top of a location infrastructure where each node can quickly (in $O(\lg n)$ steps) locate any remote node while maintaining only $O(\lg n)$ local state [22, 24, 26, 30]. This approach holds promise for scaling to distributed systems consisting of millions of participating nodes.

While existing techniques track the characteristics of a fixed set of $O(\lg n)$ nodes, a hypothesis of this work is that there are significant additional benefits from periodically distributing a different random subset of global participants to each node. By ensuring that the received subsets are uniformly representative of the entire set of participants and are frequently refreshed, nodes will eventually receive information regarding a large fraction of participants. Consider the applicability of such a mechanism to the following application classes:

- *Adaptive overlays:* A number of efforts build overlays that adapt to dynamically changing network conditions by probing peers. For instance, both Narada [14] and RON [2] maintain global group membership and periodically probe all participants to determine appropriate peering arrangements, limiting overall system scalability. The presence of a mechanism to deliver random subsets to each node would allow overlay participants to learn of remote nodes suitable for peering, while at the same time periodically learning enough new information to adapt to dynamically changing network conditions.
- *Parallel downloads:* One recent effort [5] suggests "perpendicular" downloads of popular content across a set of peers receiving erasure-coded content. Here, nodes receive data not only from the source, but also from peers that might have already

*This research is supported in part by the National Science Foundation (EIA-9972879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Albrecht is also supported by an NSF graduate fellowship and Vahdat is also supported by an NSF CAREER award (CCR-9984328).

received the data from the source or some other peer. One unresolved challenge to this approach is locating peers with both available bandwidth and diversity in the set of received data items. Random subsets would provide a convenient mechanism for locating such peers. Related to this approach, a number of efforts into reliable multicast [4] propose the use of peers in the multicast tree for data repairs (to avoid scalability issues at the root). Random subsets would likewise provide a convenient mechanism for locating nearby peers that do not share the same bottleneck link (and hence have a good chance of containing lost data).

- *Peer to peer systems:* For locality, peer to peer systems [22, 24, 26, 30] often desire multiple potential choices at each hop between source and destination. A changing, random subset of participating nodes would enable nodes to insert entries into their routing table with good locality properties and to adapt to dynamically changing network conditions.

- *Content distribution networks:* In CDNs, objects are stored at multiple sites spread across the network. Important challenges from the client perspective include resource discovery (determining which replicas store which objects) and request routing (sending the request to the replica likely to deliver the best performance given current load levels and network conditions). Random subsets would allow CDNs to track the state of a subset of global replicas. A number of earlier studies [9] indicate that making decisions based on a random subset of global information often performs comparably to maintaining global system state.

- *Epidemic algorithms:* A classic application of random subsets is epidemic algorithms [10, 27], where nodes transmit updates to random neighbors. With high probability, n nodes performing "anti-entropy" will converge to see the same set of updates in $O(\lg n)$ communication steps. Random subsets provides a convenient mechanism for locating neighbors and perhaps biasing communication to nearby sites.

Thus, we view a scalable mechanism for delivering uniformly random subsets of global participants as fundamental to a broad range of important network services. This paper presents the design and implementation of *RanSub*, one such protocol. RanSub utilizes an overlay tree to periodically distribute random subsets to overlay participants. We could leverage any number of existing techniques [3, 7, 12, 13, 14, 16, 19, 21, 23, 31] to provide this infrastructure. However, to demonstrate some of the key benefits of RanSub in support of adaptive overlay construction, we present the design and evaluation of SARO (Scalable Adaptive Randomized Overlay). SARO uses random subsets to build overlays that i) meet application-specified targets for delay and bandwidth, ii) match the characteristics of the underlying network and iii) adapt to changing network conditions.

Much like RanSub, a key goal of SARO is scalability: no node tracks the characteristics of more than $O(\lg n)$ remote participants and no node probes more than $O(\lg n)$ peers during any time period (a configurable epoch). Further, SARO requires no global coordination or locking to perform overlay transformations. A special instance of RanSub ensures a total ordering among all participants such that no two simultaneous transformations can introduce loops into the overlay.

We have completed an implementation of both SARO and RanSub and conducted a number of large-scale experiments. We show that a 1,000-node instance of SARO running on an emulated 20,000 node network using ModelNet [28] quickly converges to user-specified performance targets with low overhead from both per-node probing and RanSub operation. We further subject our prototype to live runs over the PlanetLab testbed [20], demonstrating similarly low convergence times, and the ability to stream live media over our overlays using publicly available media servers.

The remainder of this paper is organized as follows. Section 2 presents the RanSub algorithm for distributing random subsets. Section 3 then details SARO, a scalable and adaptive overlay that uses random subsets to conform to the underlying topology and dynamically adapt to changing network conditions. Section 4 evaluates our prototype's behavior under a variety of network conditions. Section 5 places our work in the context of related efforts and Section 6 presents our conclusions.

2 Random Subsets

2.1 Desirable Properties

Before we present the details of our design and implementation, we discuss desirable properties of a random subset "tool". Ideally, the system will offer:

1. *Customization:* Applications should determine the size of random subsets that are delivered. This size will depend on application-specific actions per-

formed by nodes upon receiving the random subset. For example, a parallel download application may wish to initiate data transfer with only a small constant number of peers while a P2P system may wish to probe $O(\lg n)$ nodes.

2. *Scalability:* The system should support large-scale services without posing a burden on the underlying network in terms of control overhead. Additionally, correct system operation should not depend on system size, i.e., the application should be able to request any random subset size. Overall, scalability implies that required per-node state and network communication overhead should grow sub-linearly with the number of participants.

3. *Uniform, changing subsets:* We envision a tool that is repeatedly invoked to retrieve "snapshots" of global participants at different points in time. Each snapshot, or random subset, should consist of nodes uniformly distributed across all global participants, such that each remote node appears in a delivered subset with equal probability. If desired by the appliaction, each invocation of the tool should return to each participant a *different* random subset independently chosen over all participants. Similarly, across invocations, each participant should receive probabilistically different subsets with no correlation across invocations. In this way, over time, each node can be exposed to a wide variety of global participants. Certain applications may desire non-uniform distribution that, for example, favors nearby nodes; this functionality can be layered on top of the baseline system.

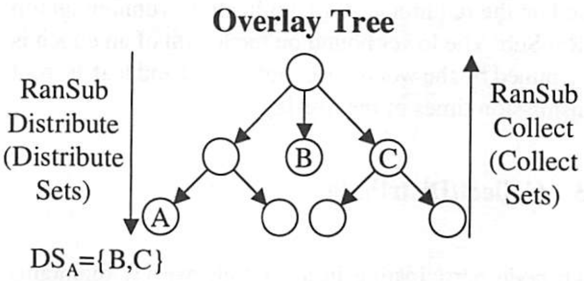
4. *Frequent updates:* To support network services that use the system to adapt to changing network conditions, the system should offer frequent distribution of random subsets.

5. *Resilience to failures:* The system will preserve its properties even in the face of failures. Failed nodes should not appear in future random subsets within a short and bounded amount of time.

6. *Resilience to security attacks:* Even when under attack by malicious users, the system should maintain its properties (uniform distribution, etc.), and degrade its performance gracefully when it is unable to defend against a massive attack.

2.2 Overview

Given the goals described above, we now describe RanSub, our scalable approach to distributing random subsets



Overlay Tree
RanSub Distribute (Distribute Sets)
RanSub Collect (Collect Sets)
A
B
C
$DS_A=\{B,C\}$

Figure 1: RanSub operation.

containing nodes that are uniformly spread across all participants. For the purposes of this discussion we assume the presence of some scalable mechanism for efficiently building and maintaining an overlay tree. A number of such techniques exist [3, 14, 16]; in the next section, we describe SARO, one technique for building such an overlay that both makes use of RanSub functionality and also provides the necessary overlay infrastructure.

Figure 1 summarizes RanSub operation. RanSub distributes random subsets through *Collect* messages that propagate up the tree and leave state at each node. *Distribute* messages traveling down the tree use soft state from the previous collect round to distribute uniformly random subsets to all participants.

RanSub distributes a subset of participants to each node once per configurable *epoch*. An epoch consists of two phases: one *distribute phase* in which data is transmitted from the root of an overlay tree to all participants (data is distributed down the tree) and a second *collect phase* where each participant successively propagates to its parent a random subset called a *collect set* (CS) containing nodes in the subtree it roots (data is aggregated up the tree). During the distribute phase, each node sends to its children a uniformly random subset called a *distribute set* (DS) of remote nodes. The contents of the distribute set are constructed using collect sets gathered during the previous collect phase.

When a Distribute message reaches a leaf in the RanSub tree, it triggers the beginning of the next collect phase where each node sends its parent a subset of its descendants (the collect set) along with other metadata. This process continues until the root of the tree is reached. The collect phase is complete once the root has received collect sets from all of its children. The root signals the beginning of a new epoch by distributing a new distribute set to each of its children, at which point the entire process begins again. The length of an epoch is configurable based on the requirements of applications running on top of RanSub. The lower bound on the length of an epoch is determined by the worst-case root-to-leaf and leaf-to-root transmission times of the overlay.

2.3 Collect/Distribute

Each node participating in a RanSub overlay maintains the following state: address of its parent in the overlay, a list of its children, and the sequence number of the current epoch. In addition, it maintains the following soft state: a collect set and number of subtree descendants for each of its children, a distribute set, and the total number of overlay participants. Below, we describe how RanSub uses this information and how it maintains it in a decentralized manner.

2.3.1 Collect Phase

Overall, the goal of the Collect message is for each node to: i) compose the collect sets for constructing the distribute set during the subsequent distribute phase, and ii) determine the total number of participants in its local subtree.

The collect phase begins at the leaves of the tree in response to the reception of a Distribute message. Table 1 describes all the fields in Collect messages (in the left half of the table). The Collect message has the same sequence number as the triggering Distribute message. At the leaves, the number of descendants is set to one and the collect set contains only the leaf node itself. Once a parent receives all Collect messages from its children, it further propagates a Collect message to its own parent. The nodes in the collect set are selected randomly from the collect sets received from its children to form a subset of configurable size ($O(\lg n)$ by default). Each node stores this collect set to aid in the construction of distribute sets distributed to its children during the subsequent distribute phase.

One key challenge is to ensure that membership in the collect set propagated by a node, A, to its parent is both random and uniformly representative of all members of the sub-tree rooted at A. To achieve this, RanSub makes use of a $Compact$ operation, which takes as input multiple subsets and the total population represented by each subset. $Compact$ outputs a new subset with two properties: i) group membership randomly chosen over the input subsets and ii) a target size constraint. This is achieved by building the output set incrementally. We first randomly choose an input subset based on the population that it represents. We then randomly choose a member of this subset (not already selected) and add it to the output set. Consider the case where $Compact$ were performed over two subsets, A and B. A and B each contain 8 members, A represents a population of 30 while B represents a population of 10. $Compact$ would choose members of A to add to the output set with probability 0.75. Thus, the output set of 8 members would uniformly represent a population of 40, with an *expected* membership of 6 members from A and two members from B. Note that $Compact$ is able to properly weigh each subset because as part of collect/distribute, each node learns the total number of nodes at the subtree rooted at each of its children.

2.3.2 Distribute Phase

A new epoch can begin once the root has received a Collect message from all of its children for the previous epoch. The actual length of an epoch is determined by individual application requirements. The right half of Table 1 describes the fields contained in the Distribute message.

A parent constructs distribute sets for each child in the following manner. Recall that each node stores the collect set received from each child during the previous collect phase. Thus, for each of k children, a particular node maintains $CS_1, CS_2, \ldots, CS_k$. Also recall that each collect set, CS_i, consists of nodes selected uniformly randomly from the subtree rooted at node i. A parent node A constructs a distribute set for each child from this information saved during the preceding collect phase. This information includes the collect set for each child, the node A itself, as well as DS_A, A's own distribute set.

To the application using it, RanSub offers three choices regarding the contents of distribute sets:

- *RanSub-all :* This is suitable when the application requires uniformly random subsets of all nodes in the system. There are two flavors of the *All* option. *All-identical* delivers the *same* distribute set to all nodes in the overlay. This distribute set, DS_{root}, is created by the root using the Compact operation:

  $$DS_{root} = Compact(CS_1, \ldots, CS_j, \{root\})$$

  where CS_i represents the subtree rooted at child i of the root (numbered $1, \ldots, j$). A potentially more useful construct is evident with the *ALL-non-identical* option that delivers *different* distribute sets

Collect		Distribute	
Sequence #	Sequence number of current epoch	*Sequence #*	Sequence number of current epoch
Collect Set	Uniformly random subset of nodes in sender's subtree	*Distribute set*	Uniformly random subset of overlay participants
Descendants	Estimate of number of nodes in sender's subtree	*Participants*	Estimate of total number of nodes in the overlay
		Reshuffle flag (only for ordered RanSub)	Determines if children should be reshuffled so that a new total ordering is created

Table 1: Contents of Collect and Distribute messages.

to each node. In this case, node Z receives a Distribute message from its parent P containing DS'_P. Z constructs DS_Z using DS'_P and the collect sets stored from its children, $CS_1, \ldots, CS_k$, in the following manner:

$$DS_Z = Compact(CS_1, \ldots, CS_k, DS'_P, \{P\})$$

Z then forwards the following to each child x:

$$DS'_Z = Compact(CS_1, \ldots, CS_{x-1}, CS_{x+1}, \ldots, CS_k, DS'_P, \{P\})$$

Note that the root's DS'_P is {} since it has no parent.

- *RanSub-nondescendants :* In this case, each node should receive a random subset consisting of all nodes except its descendants. This might be appropriate for an application-layer multicast structure where participants are probing for better bandwidth and latency to the root of the tree. In this case, considering a node's descendants could introduce a cycle in the overlay tree. For each child x (numbered $1, \ldots, k$), the parent node A constructs DS_x in the following manner:

$$DS_x = Compact(CS_1, \ldots, CS_{x-1}, CS_{x+1}, \ldots, CS_k, DS_A, \{A\})$$

- *RanSub-ordered :* This type of distribute set calculation imposes a total ordering among participating nodes. A node receives a distribute set containing random nodes that come before it in the total ordering. For each child x (numbered $1, \ldots, k$), the parent node A constructs DS_x in the following manner:

$$DS_x = Compact(CS_1, \ldots, CS_{x-1}, DS_A, \{A\}) \quad (1)$$

Our sample application, an adaptive application-layer multicast overlay, uses Ransub-ordered to ensure that simultaneous transformations to the tree structure do not introduce loops (as discussed in Section 3). Thus, we assume RanSub-ordered for the remainder of this paper.

2.3.3 Discussion

A limitation of RanSub-ordered is that the first child of a particular node will always have a smaller set of potential nodes to choose from than the kth. In fact, the first child's distribute set would always be restricted to a relatively small subset of global nodes. For RanSub-ordered, this violates our goal of distributing random subsets to *all* nodes that are uniformly chosen across all global participants in a single epoch. We take the following step to ensure that every node still receives a uniformly random subset across multiple invocations of RanSub-ordered. Every configurable r epochs, the root of the overlay periodically sets the reshuffle flag in its Distribute message, signaling overlay participants to randomly reshuffle children lists. This allows children that were at the beginning of the total ordering (and hence received few nodes in distribute sets) a chance to move toward the end of the total ordering and receive information about more nodes.

Figure 2 summarizes the operation of the two phases of the RanSub protocol. For simplicity, we do not include the results of *Compact*, which would appropriately reduce the size of all subsets to the application-specified size constraint. In the collect phase for this example, each node constructs a collect set (CS) composed of the union of itself and all members of the collect sets it received from its children. Thus, A receives a collect set from each of its children, B and C, that are uniformly representative of the subtrees rooted at B and C respectively. Each node determines which of its collect sets should be used to compose the distribute set (DS) for each of its children.

For the distribute phase, node A constructs DS_B by taking the union of itself (A) with CS_C. Node B in turn constructs DS_D by taking the union of itself (B) with its own distribute set ($DS_B = \{A, C, F, G\}$) and $CS_E = \{E\}$ from the previous collect phase. D gets "lucky" in this ordering (it is actually the last node in this total ordering)

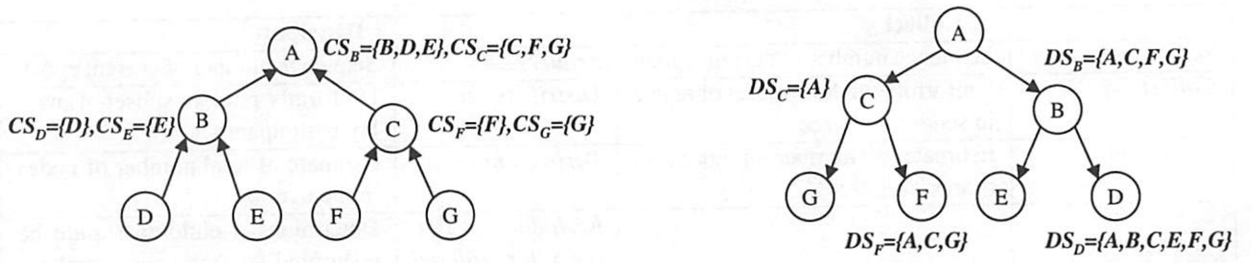


Figure 2: Example scenario depicting the two phases of the RanSub protocol: The collect phase traveling up the overlay in the left panel and the distribute phase traveling down the overlay in the right panel.

and receives a distribute set representative of the entire topology (once again, recall that we are omitting the compact operation that would throw out appropriate set elements to maintain size-constrained sets). Node *G* would get "unlucky" and only receive a distribute set consisting of itself (it is the first node in this total ordering). However, once the children lists are reshuffled, the resulting total ordering would be a random walk of the tree in which each node is only visited once yielding an entirely different new total ordering. Finally, note that the complexity of reshuffling children is only needed if a total ordering is required.

2.4 Comparison with the Ideal Random Subset Primitive

We believe that RanSub closely approximates the ideal properties outlined at the beginning of this section. Since it uses a tree to propagate sublinear-sized collect and distribute sets between parents and children, it imposes low overhead on the underlying network. Using an efficient overlay structure further ensures that epochs can be short. For instance, we find that for a 1000-node system (in a network with a diameter less than 500 ms), epochs can be as short as five seconds.

In the absence of node failure, RanSub delivers random subsets that are close to uniformly distributed. The key behind achieving uniformity is accounting for nodes that are represented by a random sample at any given time during the protocol execution. We achieve this by running the protocol over a tree, where it is straightforward for a node to have an estimate of the number of its descendants. Future work includes adapting RanSub to function over meshed overlays, not just trees.

RanSub uniformity might suffer as nodes join and leave the system. We do not provide the guarantee that nodes in the random subset will be active when considered by other nodes. RanSub is a mechanism for taking a snapshot of "live" tree participants and then taking uniformly random samples of it. This is a strength of our approach since we do not require a separate group membership mechanism. If the node is alive when it receives a Collect message, it may be included in distribute sets given to other nodes. If that node fails soon after the snapshot, it may be unavailable when considered by other nodes. If a node joins after the collect phase of a previous epoch has completed, it will not be present in the snapshot.

In RanSub, the time for node failure detection can be on the order of a small number of seconds. Nodes below a point of failure rejoin the tree at current participants using previously received distribute sets. The node failure detection interval can be further reduced if the underlying tree is used to support application-layer multicast. In this case, absence of data for a few hundreds of milliseconds might signify disconnection from the parent. In essence, application-layer data may serve as a heartbeat mechanism for failure detection.

RanSub assumes trust between overlay participants, and therefore is not resilient to internal attacks. For example, a malicious node might alter the contents of collect and distribute sets. We are investigating several techniques to address this limitation. For instance, one approach involves sending subsets over multiple tree "cross-links" to allow identification and confinement of damage caused by malicious users. These "cross-links" would also help with the overall reliability of the tree. Multiple paths through the tree means that a single failure may not disconnect nodes in the tree.

3 SARO

3.1 Overview

As discussed earlier, distributing uniformly random subsets of global participants is applicable to a broad range of important services. This section describes SARO, a Scalable Adaptive Randomized Overlay, one such application of RanSub. The use of RanSub for SARO is circular in this example. SARO uses random subsets to probe peers to locate neighbors that meet performance targets and to adapt to dynamically changing network conditions. At the same time, RanSub uses the SARO overlay for efficient distribution of Collect and Distribute messages.

The goal of SARO is to construct overlays that are: i) scalable, ii) degree-constrained, iii) delay- and bandwidth-constrained, iv) adaptive, and v) self-organizing. For scalability, we enforce the following rules:

1. No node should track more than $O(\lg n)$ remote nodes.
2. No node should perform more than $O(\lg n)$ network probes during any time period (epoch). The application can configure the number of probed nodes to effect a tradeoff between network overhead and the adaptivity (or agility [17]) of the overlay.
3. No global locking should be required to transform the overlay.

We achieve the first two goals using the distribute sets that RanSub transmits to each node every epoch. Each SARO node A performs probes to members of this subset to determine if a remote node B exists that would deliver better delay or bandwidth to A and its descendants. If so, A attempts to move under B.

To motivate the third requirement, consider a node A that decides to move underneath a remote node B. The system would introduce a loop if some ancestor of B simultaneously decides to move under some descendant of A. The naive approach to avoiding loops requires locking a number of nodes across the wide area to avoid such simultaneous overlay transformations. While this may be appropriate for a small number of nodes or for LAN settings, this process will not scale to large overlays. Thus, we impose a total ordering among nodes provided by the ordered flavor of RanSub to ensure that no two simultaneous moves in the same epoch can introduce a SARO loop. During any epoch, each node's distribute set contains only remote nodes that come before it in the current total order.

Recall that RanSub periodically changes the total ordering of nodes and random membership in delivered distribute sets. Thus, while each node only tracks and probes $O(\lg n)$ remote nodes during any one epoch, RanSub ensures that the makeup of the distribute set changes probabilistically such that, over time, each node quickly probes all potential parents. The size of the random subset effects a tradeoff between scalability (measured by per-node state and network probing overhead) and convergence time (the amount of time it takes to build an overlay that achieves delay and bandwidth targets even under changing network conditions).

SARO requires additional information beyond that distributed by RanSub (see Table 1). In general, applications may wish to piggyback different state information with the existing Collect and Distribute messages. Our RanSub layer is designed in a manner to make such extensibility straightforward, though a detailed description is beyond the scope of this paper. Table 2 describes the additional information transmitted by SARO in Collect and Distribute messages.

3.2 Probing and Overlay Transformations

Recall that the length of an epoch (random subsets are distributed once per epoch) is determined by application-specific requirements. Shorter epochs provide more information, while longer epochs incur less overhead. For SARO, our goal is to quickly converge to an overlay that matches the underlying topology and adapts to dynamically changing network characteristics. The implementation and evaluation in this paper is pessimistic in that we run with a constant epoch length of 10 seconds in all cases. In the future, we plan to investigate setting the epoch length adaptively based on both overlay characteristics and network conditions. For example, if SARO has already matched the underlying topology and if network characteristics are not changing rapidly (likely the common case in the Internet), then the system can afford a longer epoch length. Thus, we envision reducing overall system overhead by running SARO with a short epoch length during initial self-organization and in response to large changes in network conditions, but running with a long epoch in the common case.

A key to SARO's ability to converge to bandwidth and delay (maximum delay from root to all other overlay participants) targets lies in localized tree *transformations*. Dur-

Collect		Distribute	
Delay	Delay estimate for furthest descendant	*Tree height*	Estimate of the actual highest root-to-leaf latency in the tree
Delay gain	Estimate of delay gain from moving to a best alternative parent	*Root delay*	Estimate of recipient's delay from root

Table 2: Additional fields in Collect and Distribute messages required by SARO.

ing each epoch, nodes measure the delay and bandwidth between themselves and all members of their distribute set. These probes consist of a small number of packets inter-spaced by the target application bandwidth[1]. If the loss rate of the probes is below a specified threshold, the probing node calculates the average round trip time. The goal is to locate a new parent that will deliver better delay, better bandwidth, or both to itself and all of its descendants. If a better parent is located, the child attempts to move under it. The migrating node issues the add request to the potential parent and waits for the response. If the request is accepted, it notifies its old parent, communicates its new delay from root to all its children, and notifies the new parent of its furthest descendant (updating the parent's state for the next epoch).

In general, the goal of SARO is to achieve the lowest delay configuration that still maintains the target bandwidth to the root. Thus, during each epoch, nodes use information from their probes to perform two types of transformations: `BANDWIDTH_ONLY` and `DELAY_AND_BANDWIDTH`. Nodes that have not yet reached their bandwidth target may perform `BANDWIDTH_ONLY` transformation to improve their bandwidth to root, even if it means increasing their delay. `DELAY_AND_BANDWIDTH` transformations allow nodes to rotate under a new parent that improves the nodes' delay to root while maintaining (or improving) bandwidth.

3.3 Dynamic Node Addition and Failure Recovery

To this point, our discussion assumes a static set of nodes dynamically self-configuring to match changing network conditions. In general however, the set of overlay participants will also be changing. A node performing a SARO join simply needs to contact any existing member of the overlay. The initial bootstrapping parent may be sub-optimal from the perspective of bandwidth or delay. However, the node will begin receiving random subsets as part of the collect/distribute process, and it can use the associated random subsets to probe for superior parents using the process described above.

Since the bootstrapping parent node might fail, we make an additional assumption that every node is aware of the root of the tree. Therefore, an incoming node will always have at least one bootstrapping parent (similar to Overcast [16], we can replicate the root to improve root availability). For the sake of scalability, we allow nodes to contact the root only if it is absolutely necessary (i.e., its original bootstrapping parent failed).

If a bootstrapping parent does not have enough slots in its children list to accept a joining node, it redirects the incoming node randomly to either its parent or one of its children. Similarly, if the incoming node would violate the delay bound, the bootstrapping parent redirects it to its parent. If the node is redirected too many times, it joins the tree at the first available point, and tries to improve its position later.

Handling node failure is simplified by our periodic distribution of Collect and Distribute messages, which implicitly act as heartbeat messages. Each parent waits for Collect messages from all of its children. If a message is not received within some multiple of the subtree height[2], the parent assumes that one of its children has failed and excludes it from participating in the next epoch by not sending a Distribute message to that child. However, the node does proceed with a Collect message to its own parent when it detects the failure to ensure that a failure low in the tree does not cascade all the way back up the tree. A node, A, can similarly detect the failure of its parent when it does not receive a Distribute message within a multiple of the delay target. In this case, A will send a dummy Distribute message (with an empty distribute set) to all of its children. This empty Distribute message signals A's descendants that no probing or overlay transformations should be performed during this epoch. A then attempts to locate a new parent using information from

[1]We currently assume that overlay traffic makes up a relatively small portion of overall traffic through the bottleneck. We leave more accurate, more stable, and TCP-friendly probing to future work. In general, higher accuracy probes will inherently incur higher overhead, though this issue is orthogonal to our own work.

[2]Note that the current maximum node-to-leaf delay serves as a convenient baseline for the complex process of determining appropriate timeouts.

previous distribute sets where appropriate. Thus, upon a failure, the entire subtree rooted at A is able to rejoin the overlay with a single transformation, rather than forcing all nodes below A to rejoin separately.

3.4 Weans

Under certain circumstances, the greedy nature of SARO can lead to sub-optimal overlays. Consider the following situation. A node A with a particular degree bound has a full complement of children. However, a node B somewhere else in the overlay can only achieve its bandwidth target by becoming a child of A. Finally, one of A's children, C, is best served by A but would still be able to achieve its bandwidth target as a child of some fourth node D. As described thus far, SARO will become stuck in a "local minimum" in this situation.

We address this situation by introducing *wean* operations. At a high level, the goal is to ensure that each parent leaves a number of slots open whenever possible to address the above condition. Thus, as a node approaches its degree limit, it will send a wean message to one of its children. In subsequent epochs, that child will move to a new parent if it can find a suitable location that still meets its bandwidth requirement (though its delay may be increased). A wean may or may not succeed (an appropriate alternate parent may not exist) and the wean operation expires after a configurable number of epochs.

One difficulty is choosing the child to wean. Ideally, a parent would wean the child that would lose the least in delay and bandwidth while still achieving its targets. We approximate this in the following manner. During each epoch, all nodes maintain information on the *best alternate parent* with respect to both delay and bandwidth. This information is propagated to parents in the collect phase (see Table 2) and is used by parents to determine the wean target.

4 Evaluation

We have completed an implementation of both RanSub and SARO as described in the previous sections. We wrote our code to a compatibility layer that allows us to evaluate our working system in both the *ns* packet simulation environment [18] and across live networks. For brevity, we omit the majority of the results of our detailed simulation evaluation. Instead, we focus on the behavior of our system running live on ModelNet [28], an Internet emulation environment and across the Internet in the PlanetLab testbed [20].

For our ModelNet experiments, SARO runs on 35 1.4Ghz Pentium-III's running Linux 2.4.18 and interconnected with both 100 Mbps and 1 Gbps Ethernet switches. We multiplex 28-29 instances of SARO on each of the Linux nodes, for a total of one thousand nodes self-organizing to form overlays. We validated our ModelNet results with ns experiments using identical topologies and communication patterns. In ModelNet, all packet transmissions are routed through a *core* responsible for emulating the hop-by-hop delay, bandwidth, and congestion of a target network topology. For our experiments, we use a single 1Ghz Pentium III running FreeBSD-4.5 as the core. The core has a 1 Gbps connection to the rest of the cluster. The core was never a bottleneck either in CPU or bandwidth for any of our experiments. Our earlier work [28] shows that, for a given wide-area topology, ModelNet is accurate to within 1 ms of the target end-to-end packet transmission time up to and including 100% CPU utilization, and 120,000 packets/sec (1 Gbps assuming packets are 1000 bytes on average). ModelNet can scale its capacity with additional core nodes, though this was not necessary for our experiments here. ModelNet emulates packet transmission hop-by-hop (including per-hop queue sizes and queuing disciplines) through a target network. Thus, a packet's end-to-end delay accounts for congestion and for per-hop queuing, propagation, and transmission delay.

By default, the core emulates the characteristics of a random 20,000-node INET-generated topology [6]. We randomly assign 1,000 nodes to act as clients connected to one-degree stub nodes in the topology. One of these participants is randomly selected to act as the root of the SARO tree. We classify network links as being Client-Stub, Stub-Stub (both with bandwidth between 1-10 Mbps), Transit-Stub (bandwidth between 10-100 Mbps) and Transit-Transit (100-155 Mbps). We calculate propagation delays among nodes from the relative placement of the nodes in the plane by INET. The baseline diameter of the network is approximately 200 ms. While the presented results are restricted to a single topology, the results of additional experiments and simulations all show qualitatively similar behavior.

4.1 RanSub Uniformity

To verify that RanSub distributes uniformly random subsets to all nodes, we used our complete RanSub prototype to create a SARO overlay of 1000 emulated nodes as described above. After convergence, we let our experiment run for a total of 360 epochs and tracked the cumulative number of unique remote peers that RanSub distributes to each node over time. We configured RanSub to distribute

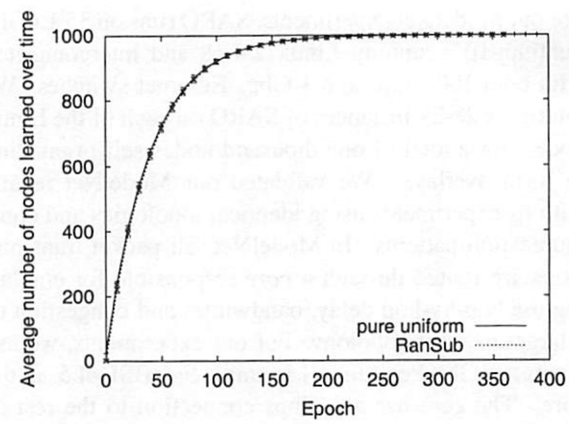


Figure 3: Average number of nodes each of 1,000 nodes learns of as a function of number of epochs for the optimal pure uniform case and for RanSub.

25 random participants each epoch. Figure 3 plots the average number of peers each node is exposed to on the y-axis as a function of time progressing in epochs on the x-axis. The vertical bars represent the standard deviation. We also show the best case in which we simulate pure uniform random subsets (using the same random number generator as the one used by our RanSub implementation). RanSub delivers random subsets with essentially optimal uniformity.

4.2 SARO Overlay Convergence

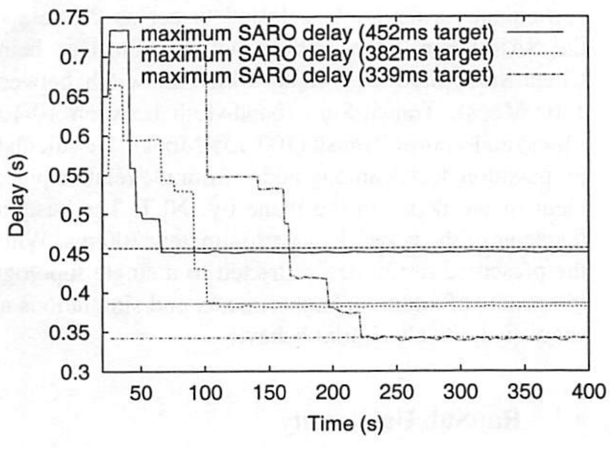


Figure 4: Delay convergence as a function of time for three different delay targets.

Figure 4 shows the convergence time of three SARO overlays running with a maximum degree of 10 and random subsets of size 15. The 1000 nodes join the overlay sequentially at a random point in the network over the first 20 seconds of the experiment (50 nodes/second) and then use random subsets to probe for parents that will deliver the appropriate delay target (bandwidth targets of 64Kbps were easily achieved for this experiment). In effect, at the beginning of the experiment we pessimistically create an overlay with random interconnectivity. We observe the behavior of the system for three different delay targets, 339 ms, 382 ms, and 452 ms. The 339 ms delay target is quite difficult to achieve for our topology and degree bound. The figure plots the achieved worst-case delay relative to the delay target as a function of time progressing on the x axis. SARO uses random subsets to converge to the specified delay target in all cases, with convergence time varying from 60 seconds to 220 seconds in the three cases depending on the tightness of the delay target. We note that our convergence times using random subsets are comparable to a number of smaller-scale overlay construction techniques that maintain global state information and that perform global probing.

4.3 Effects of Random Subset Size

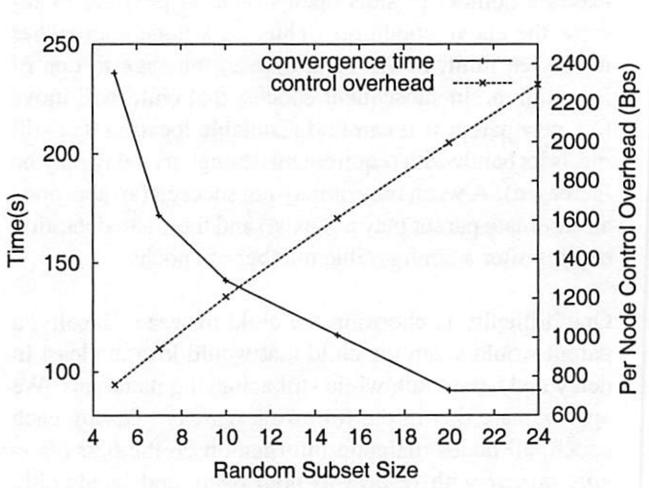


Figure 5: Delay convergence time and resulting per-node probing overhead as a function of the size of the random subset.

We now quantify the effects of the size of the random subsets on SARO convergence time. In general, less information in the random subset increases convergence time as nodes have to spend more time to find an appropriate parent. More information will likely decrease the convergence time, but at the cost of increased network probing overhead. With a maximum per-node degree of 10, we measure the time for the 1,000 node SARO tree to converge to a 382 ms delay target as a function of the size of the random subsets. As shown in Figure 5, we increase the size of the random subset from 5 to 24 on the x-axis and plot the resulting convergence time on the left-hand y-axis. The convergence time decreases from approximately 240 seconds to 90 seconds. Figure 5 also plots the associated tradeoff with per-node control overhead, accounting for both RanSub Collect/Distribute messages and probing overhead. As expected, probing overhead on the right-hand y-axis grows linearly with the size of the random subset, but only to a manageable 2300 bytes/sec even when the random subset size grows to 24, most of which is probing overhead. Note that the benefits of increasing the random subset size beyond 20 diminishes rapidly. Of course, this point of diminishing returns will vary with the topology and delay target. We have experimented with dynamically increasing or decreasing the random subset size by taking real-time measurements of the overlay's convergence, but we leave this to future work.

4.4 Adaptivity

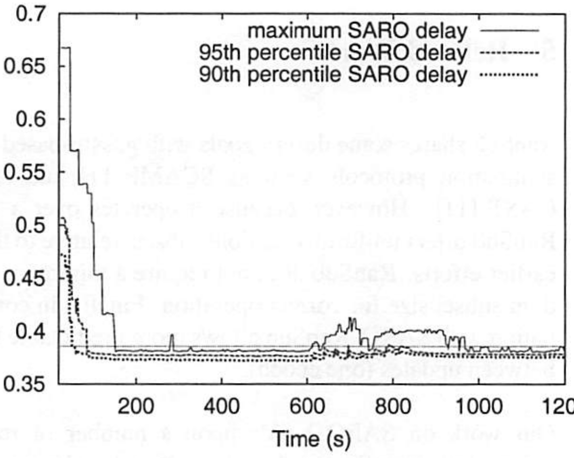


Figure 6: Adaptivity of a SARO tree in response to pronounced change in network delay.

One of the most important aspects of SARO is its ability to dynamically react to changing network conditions. To evaluate this ability, we subject a steady-state 1000-node SARO tree to widespread and sustained change in network characteristics. Every 25 seconds, we increase the propagation delay of a randomly chosen 14% of all network links by between 0-25% of the link's original delay. The idea behind this experiment is to determine what happens to the overlay as the network continuously degrades under conditions much worse than those typically found on the Internet. Figure 6 shows the results of this experiment, plotting delay as a function of time for the 90th percentile, 95th percentile, and worst case node in the overlay as a function of time progressing the x-axis. Note that the 90th percentile indicates that 90% of SARO nodes have delay better than the indicated value. We set the delay target to 382 ms for this experiment, the degree bound to 10 and the size of the random subset to 15.

We intentionally set our target delay to a value that is relatively difficult to achieve for our degree bound in the face of highly variable network conditions. Thus, the overlay initially takes approximately 150 seconds to converge to the delay target. We perturb network conditions in the manner described above beginning at time $t = 600$ and continuing for 200 seconds. While 95% of nodes are able to maintain their delay target during most of the network perturbation, it takes SARO another 180 seconds after the network perturbation subsides (though link delays remain at their elevated levels) to once again bring all nodes within delay bounds. We include the results of this experiment to quantify the level of adaptivity that SARO can deliver. Additional experiments indicate that if network conditions were perturbed for a longer period of time (with same number of links, magnitude of change, and frequency of change), SARO would be unable to once again achieve the delay target. With less perturbation or a more relaxed delay target, SARO typically quickly recovers from changes to network conditions.

4.5 PlanetLab Deployment

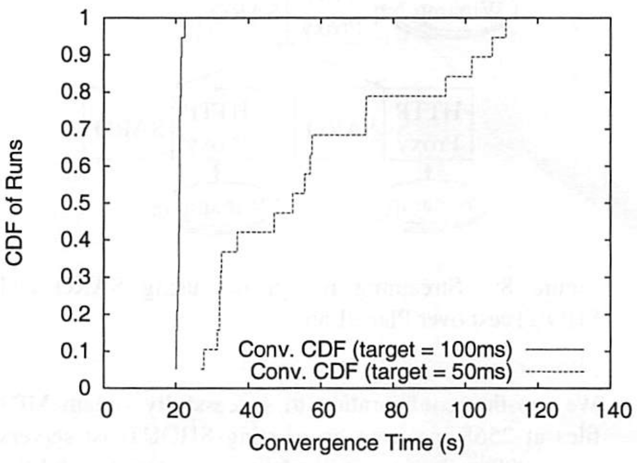


Figure 7: CDF of convergence time for 19 PlanetLab nodes, with each node acting as root in turn for two different delay targets.

To further evaluate the utility of our approach, we evaluated the behavior of SARO running on a subset of 19 PlanetLab nodes [20] during September 2002. We ran SARO over PlanetLab 19 times with each separate run using a different PlanetLab node as the root of the overlay. We then measure the convergence time for each of two different delay targets: 50ms and 100ms. We set the maximum per-node degree to 5 and the random subset size to 5. Figure 7 plots the result of this experiment. We find that for a relatively relaxed delay target of 100ms, all nodes converge within 20 seconds. Tightening the delay bound to 50 ms increases the typical convergence time to approximately a minute with worst case convergence taking up to two minutes. Note that all reported values are once again for the worst-case convergence time of the last node to meet its delay target.

Next, we streamed live audio over SARO by integrating an HTTP proxy as a separate thread in the SARO address space, as depicted in Figure 8. We use a publicly available SHOUTCast server [25], to stream MP3 encoded over HTTP. We then instantiate a SARO/HTTP process on the same node as the SHOUTCast server to act as the root of the overlay tree. Each SARO node below the root instructs its associated HTTP proxy to establish a connection to its parent to receive streaming data. The node is then able to stream content to local Winamp players and to its own children in the overlay. Each time a SARO node locates a better parent, it also instructs is HTTP proxy to reestablish a connection to this new parent.

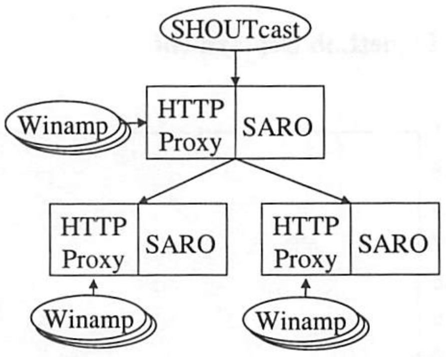


Figure 8: Streaming live media using SARO and SHOUTcast over PlanetLab.

We use this configuration to successfully stream MP3 files at 256Kbps from an existing SHOUTcast servers over 1000 nodes in our emulation environment and the PlanetLab testbed. Figure 9 shows the results of a 5-minute experiment of SHOUTcast streaming over 19 SARO nodes spread across the PlanetLab testbed. We set the delay target to 75ms, the size of the random subsets to 5 and the maximum per-node degree to 5. The figure plots a CDF of the percent of bytes received by each of the 19 nodes. Byte loss rates vary from 0-5%. However, we note that we measure the loss rate while SARO was still self-organizing at the beginning of the experiment. The vast majority of the losses came at this time. We verified the correctness of our experiment by connecting to individual HTTP/SARO nodes using the Winamp media player to playback the stream.

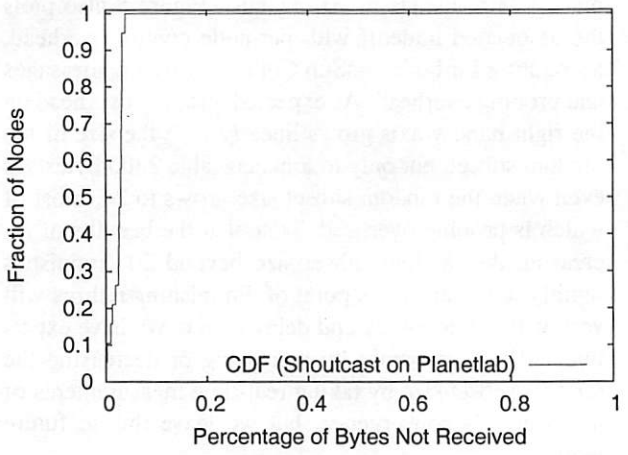


Figure 9: Distribution of percent of packets received for a 5 minute experiment streaming 256Kbps over SARO running on PlanetLab.

5 Related Work

RanSub shares some design goals with gossip-based dissemination protocols such as SCAMP [1] and LBPCAST [11]. However, because it operates over a tree, RanSub offers uniformly random subsets relative to these earlier efforts. RanSub does not require a minimum random subset size for correct operation. Finally, in combination with SARO, RanSub allows more predictable time between updates (one epoch).

Our work on SARO builds upon a number of recent efforts into "application-layer" multicast, where nodes spread across the Internet cooperate to deliver content to end hosts. Edges in this overlay are TCP connections, ensuring congestion control and reliability in a hop-by-hop manner. Perhaps most closely related to our effort in this space is Narada [14, 15], which builds a mesh interconnecting all participating nodes and then runs a standard routing protocol on top of the overlay mesh. Relative to our work, Narada nodes maintain global knowledge about all group participants. In comparison, we use the RanSub layer to maintain information about a probabilistic $O(\lg n)$ subset of global participants, making applications built on top of RanSub, including SARO, more scalable.

SARO bears some similarity to the Banana Tree Protocol (BTP) [13], and Host Multicast Tree Protocol (HMTP) [29]. However, neither of these approaches attempt to provide delay or bandwidth guarantees and neither considers a two-metric network design. All three protocols use the idea of tree transformations based on local knowledge (obtained through limited network probing) to improve overall tree quality. However, BTP implements a more restrictive policy for choosing a potential parent, called "switch-one-hop", which considers only grandparents and immediate siblings. HMTP can introduce loops and thus requires loop detection that requires knowledge of all of one's ancestors. Since HMTP offers no bounds on tree height, message size and required state are also unbounded ($O(n)$), rendering this approach potentially unscalable. Finally, HMTP is not evaluated under changing network conditions.

Yoid [12] shares the design philosophy that a tree can be built directly among participating nodes without the need to first build an underlying mesh. Yoid does not describe any scalable mechanism for conforming to the topology of the underlying network, has not been subjected to a detailed performance evaluation, and contains loop detection code as opposed to our approach of avoiding loops.

ALMI [19] uses all-pairs probing at a cost of $O(n^2)$ and transmits this changing connectivity information to a centralized node that calculates an appropriate topology for the overlay. RMX [7] faces similar scalability limitations. In Overcast [16], all nodes join at the root and migrate down to the point in the tree where they are still able to maintain some minimum level of bandwidth. Relative to our effort, Overcast does not focus on providing delay guarantees (given its focus on bandwidth-intensive applications). Its convergence time is also limited by probes to immediate siblings and ancestors. NICE [3] uses hierarchical clustering to build overlays that match the underlying network topology. Relative to our approach, NICE focuses on low-bandwidth (i.e., single-metric, delay-optimized) applications and requires loop detection code. We believe that a variety of existing overlay construction techniques, including Yoid, ALMI, RMX, Overcast, and NICE could benefit from the availability of our RanSub layer.

Finally, a number of recent efforts [21, 23, 31] propose building application-layer multicast on top of scalable peer-to-peer lookup infrastructures [8, 22, 24, 30]. While these projects demonstrate that it is possible to probabilistically achieve good delay relative to native IP multicast, they are unable to provide any performance bounds because of the probabilistic nature of the underlying peer-to-peer system. Further, these systems do not focus on two-metric network optimization (e.g., delay and bandwidth). Finally, to the best of our knowledge, these approaches, have largely been evaluated through simulation and have not been subjected to live Internet conditions.

6 Conclusions

This paper argues for a generalized mechanism for periodically distributing state about random subsets of global participants in large-scale network services. Sample applications include epidemic algorithms, reliable multicast, adaptive overlays, content distribution networks, and peer-to-peer systems. This paper makes the following contributions:

- We present the design and implementation of a scalable protocol, RanSub, that distributes state about uniformly random subsets of configurable size once per application-specific epoch.
- We argue for the utility and generality of such an infrastructure through the evaluation of SARO, a scalable and adaptive overlay construction protocol. SARO is able to match underlying networking topology and to adapt to dynamically changing network conditions by sampling members of its random subset once per configurable epoch.
- A large-scale evaluation of 1000 SARO nodes in an emulated 20,000 node network topology confirm the scalability and adaptivity of our approach.
- We use an existing streaming media player to transmit live streaming media using SARO running on top of the PlanetLab Internet testbed.

Acknowledgments

We thank our shepherd, Steve Gribble, for helping us refine the paper and David Becker for his help with ModelNet. We also thank Rebecca Braynard and anonymous reviewers for their helpful comments.

References

[1] A.J.Ganesh, A.-M Kermarrec, and L. Massoulie. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International workshop on Networked Group Communication*, 2001.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, October 2001.

[3] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, August 2002.

[4] Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transaction on Computer Systems*, 17(2), May 1999.

[5] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *Proceedings of ACM SIGCOMM*, August 2002.

[6] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[7] Yatin Chawathe, Steven McCanne, and Eric A. Brewer. RMX: Reliable multicast for heterogeneous networks. In *INFOCOM (2)*, pages 795–804, 2000.

[8] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[9] Michael Dahlin. Interpreting Stale Load Information. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1999.

[10] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic Algorithms for Replicated Database Maintenance. *Operating Systems Review*, 22(1):8–32, 1988.

[11] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, 2001.

[12] Paul Francis. Yoid: Extending the Internet Multicast Architecture. Technical report, ICSI Center for Internet Research, April 2000.

[13] David A. Helder and Sugih Jamin. End-host multicast communication using switchtrees protocols. In *Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, 2002.

[14] Yang hua Chu, Sanjay Rao, and Hui Zhang. A Case For End System Multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[15] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.

[16] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

[17] Brian Noble, M. Satyananarayanan, Giao Nguyen, and Randy Katz. Trace-based Mobile Network Emulation. In *Proceedings of SIGCOMM*, September 1997.

[18] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[19] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[20] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[21] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Third International Workshop on Networked Group Communication*, November 2001.

[22] Sylvia Ratnasamy, Paul Francis Mark Handley, Richard Karp, and Scott Shenker. A Content Addressable Network. In *Proceedings of SIGCOMM 2001*, August 2001.

[23] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[25] SHOUTcast. `http://www.shoutcast.com`.

[26] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

[27] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[28] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[29] Beichuan Zhang, Sugih Jamin, and Lixia Zhang. Host Multicast: A Framework for Delivering Multicast To End Users. In *Proceedings of INFOCOM*, 2002.

[30] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[31] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.

ISBN 1-931971-08-0